Syncpal: A Simple and Iterative Reconciliation Algorithm for File Synchronizers

Marius Shekow[✉]

Fraunhofer FIT, Sankt Augustin, Germany
marius.shekow@fit.fraunhofer.de

Abstract. Today file synchronizers are tools often used to facilitate collaboration scenarios and data management across multiple devices. They replicate the file system, e.g. from a cloud storage to a device disk, achieving convergence by only transmitting detected changes. A popular variant available in a plethora of products are state-based file synchronizers such as the Dropbox client. They detect operations by computing the difference between a previously persisted state and the respective current state. However, state-based synchronization is difficult because we need to detect and resolve conflicting operations as well as the propagation order of non-conflicting operations. This work presents Syncpal, an algorithm that reconciles two divergent file systems using an *iterative* approach. It first handles conflicts, one at a time, making sure that resolving one conflict does not negatively affect other ones, while avoiding conflicts whenever possible. It then finds order dependencies (and breaks cycles) between the remaining non-conflicting operations to avoid the violation of operation preconditions during propagation. This work is relevant for file synchronizer researchers and developers who want to improve their products with an algorithm whose iterative nature reduces the overall complexity and the probability of bugs. In addition to our proposed algorithm and a formal analysis of the underlying problem, our validation approach for the proposed algorithm includes the presentation of a full-scale implementation of an exemplary file system model.

Keywords: File synchronizer · File system · Optimistic replication · Conflict detection

1 Introduction

Today tools like word processors are a core component in digital workflows. They are used to create large parts of the user's data in the form of files, which are stored and distributed on multiple devices in a hierarchical *file system*. However, copying files and directories between storages causes various problems, both for individual users and collaborative settings. For instance, users may fail to locate

J. Pereira and L. Ricci (Eds.): DAIS 2019, LNCS 11534, pp. 1–18, 2019.
https://doi.org/10.1007/978-3-030-22496-7_1

the correct, up to date version of a document on the right device [5,8,22], and files are prone to lose their context and meta-data information when transferred via Email or instant messaging [25]. One convenient solution for such challenges is *data synchronization. File synchronizers* [2] are synchronizers whose data is the file system, including its namespace structure and file contents. They provide *optimistic replication* to otherwise isolated file systems, with weak, *eventual consistency* [21] guarantees. In particular, *cloud storage* file synchronizers like Dropbox, Google Backup and Sync, OneDrive, ownCloud and others[1] have become popular over the last ten years, indicated by the high number of their users [9,18]. They synchronize *two* file system replicas—a directory on the local storage of a device, and a directory on a cloud storage server, in near real-time. As they are not integrated on a kernel-level with the operating system, they use a *state-based* approach that detects operations by computing the difference between the current and a persisted file system state.

When using synchronizers files are available on the local disk, thus users can work *offline* for extended time periods (e.g. while traveling). The side effect is an increased chance for conflicting operations as well as long, non-conflicting operation sequences resulting from users reorganizing the folder hierarchies. These are challenging to detect and propagate for the synchronizer. For example, a conflict situation, where the user creates a new file at path '/dir/file' but '/dir' was already deleted on the server, must be detected and resolved in favor of one of the operations. But even non-conflicting operations can be challenging to propagate. Consider the situation where the user swaps two objects at paths '/x' and '/y' on the local disk, using three *rename* operations. The synchronizer's state-based update detection mechanism detects them as *two* operations (*move('/x', '/y')+move('/y', '/x')*. If the corresponding objects were not modified on the server since the last synchronization (which makes the operations *non*-conflicting), the synchronizer cannot apply these two detected *move* operations to the server because they would violate the move operation's precondition that requires the target location to be free.

Contrary to the marketing materials of industrial synchronizers which promise that their product "just works", we observed that they misbehave and make intransparent decisions for the user - especially when attempting to synchronize after a long offline phase. This includes:

- Inexplicable changes made to the file system, convoluting its structure, e.g. with file and folder (conflict) copies where no conflict actually happened,
- Ineffective use of network bandwidth, in particular when *move* operations were not detected correctly in replica X, causing the synchronizer to retransmit large files as new, rather than moving them on replica Y,
- Bugs or crashes of the file synchronizer, resulting in permanently divergent replicas, or even data loss.

All these problems cause frustration because users then have to repair directory structures and file contents manually. The majority of issues can be traced

[1] E.g. Amazon Drive, Box, NextCloud, Seafile, SpiderOakOne, LeitzCloud, Tonido, TeamDrive, Strato HiDrive, or Hubic.

back to an incomplete analysis of the underlying file system model (and its operations) by the synchronizer authors. In this work we contribute *Syncpal*, a generic algorithm for file synchronizers that eliminates above side effects, because it provides a simple and iterative solution to solving conflicts and propagating non-conflicting operations. It is based on a formally defined file system model, which makes its individual steps provably correct. Additionally, it is able to avoid conflicts whenever possible, resolves conflicts without side effects for other conflicts, and does not replace detected *move* operations with *delete* and *create* operations. This improves propagation performance, preserves meta-data (which would otherwise be lost due to the *delete* operation) and maintains usability, because users will be able to identify the move operations of their own replica in the respective operations log of the other replica [20].

We start with providing background on file synchronizers, file systems and state-based update detection in Sect. 2. After presenting the generic approach in Sect. 3 we apply it to a concrete file system model in Sect. 4. We briefly present the evaluation of an implementation of our approach in Sect. 5 and conclude in Sect. 6.

2 Background

We begin with a short introduction to file synchronizers in Sect. 2.1. As file systems are the core component being synchronized, we briefly explain differences in how file systems can be modeled and formally present our own, exemplary model in Sect. 2.2, which we use in the remainder of this work. In Sect. 2.3 we briefly explain how operations are detected in a state-based approach.

2.1 File Synchronizers

In [2] the authors describe and coin the term *file synchronizer* as a user-invoked program that performs a pair-wise synchronization of *two* file system replicas. They describe a *state-based* approach [21] with three stages, *update detection*, *reconciliation* and *propagation*. In contrast, *operation-based* approaches like [13,15] rely on a complete log of operations. Because some file systems (e.g. POSIX APIs) do not provide such logs, it is reasonable to assume that cloud storage synchronizers (and other products) use a state-based approach with a similar three-stage process. State-based approaches persist the file system state (structure + meta-data) in a database and compute operations by comparing the persisted and current state, see Sect. 2.3 for more details. Surprisingly, while there is a plethora of file synchronizer products, the topic has not received much attention in comparison within academia (neither state- nor operation-based synchronizers).

2.2 File System Model

Every file synchronizer uses its own internal file system model definition for the state. An analysis of related works reveals several differences:

- *Identity- vs. path-based model*: as discussed in [23, section 3] the file system and its operations can be modeled using the *identity*-based approach where each object is identified by a unique ID, or by a *path*-based approach where objects are only identified by their path. ID-based approaches include [3,10–13,23], for path-based approaches see [2,4,15,24].
- *Hardlink* support for files: an identity-based model may support that a specific *file* is linked exactly once, or several times. In the latter case a file's name may be part of the file itself, or be part of the parent-child link.
- *Directory* support: Most file system implementations support directories. However, alternatives exist, e.g. models that only consist of a set of *file* paths and their identities [19, Definition 2.3.1 + section 2.4.4]. Another example is Git [24] which does not support *empty* directories.
- *Operation* support: while the models of all file synchronizers we examined support *create directory*, *create file* and *edit* operations (that update the content of a file), support for other operations varies. For example, the model may or may not offer a *move* operation, or the *delete* operation may be modeled as such, or as a *move* operation to the garbage directory [13].

Because there may be a mismatch between the internal model definition and the definitions of the two underlying replicas being synchronized, file synchronizers belong to the category of *heterogeneous* synchronization [1,6,17].

We now present a formal file system model that is used in the remainder of this work. It is ID-based, because the file systems industrial synchronizers are ID-based, too, and because IDs allow to efficiently detect *moved* objects.

We define the file system $\mathcal{F}$ to be a set of tuples where each tuple represents an object with a unique ID $i \in I$, parent directory ID $p \in I$, type $t \in T$ (with $T = \{\mathit{file}, \mathit{dir}\}$), name $n \in \Sigma^+$ (with $\Sigma^+ = \Sigma^* \backslash \{\epsilon\}$), *lastmodified* meta-datum $l \in L$ and content $b \in B$. I is the set of unique IDs, L is the set of all valid lastmodified meta-datum values (e.g. $\mathbb{N}$ or arbitrary strings), and B is the set of arbitrary byte sequences, including ϵ. That is, $\mathcal{F} \subset I \times I \times T \times \Sigma^+ \times L \times B$, with tuples $(i_k, p_k, t_k, n_k, l_k, b_k)$ with $t_k = \mathit{dir} \implies b_k = \epsilon$. Several invariants hold for $\mathcal{F}$:

$$\forall i, j \in I : i \in \mathit{list}(j) \implies \mathit{type}(j) = \mathit{dir} \tag{1}$$

$$\forall i \in I : i \notin \mathit{list}(i) \tag{2}$$

$$\forall i, j, k \in I : j \neq k \land i \in \mathit{list}(j) \implies i \notin \mathit{list}(k) \tag{3}$$

$$\forall i \in I : i_{root} \notin \mathit{list}(i) \tag{4}$$

$$\forall i \in I \setminus \{i_{root}\} : \mathit{type}(i) \neq \mathit{error} \iff \mathit{ancestor}(i_{root}, i) \tag{5}$$

$$\forall i, j, k \in I : j \neq k \land j \in \mathit{list}(i) \land k \in \mathit{list}(i) \implies \mathit{name}(j) \neq \mathit{name}(k) \tag{6}$$

where $\mathit{list}(i)$ returns the set of IDs of all tuples whose $p_k = i$ (i.e., the set of immediate child IDs of i); $\mathit{type}(i)$ returns t_k of the tuple where $i_k = i$, or *error* if no such tuple exists; $name(i)$ returns n_k of the tuple where $i_k = i$. We additionally define the predicate

$$ancestor(i, j) = \begin{cases} true & j \in list(i) \\ true & \exists k \in list(i) : ancestor(k, j) \\ false & \text{otherwise} \end{cases}$$

to express whether the object with ID i is an ancestor of the object with ID j. $\mathcal{F}$ is an arborescence rooted in the well-known object $i_{root} \in I$ with $type(i_{root}) = dir$, where each object exists exactly once.

The operations with their pre- and postconditions are defined in Table 1. Function $id(i, n)$ returns the ID of the object with parent i and name n, or *error* if no such object exists. $lastmodified(i)$ returns l_k of the tuple where $i_k = i$, or *error* if no such tuple exists. $content(i)$ returns b_k of the tuple where $i_k = i$, or *error*.

We refer to [14, Section 8.5] for an equivalent formal definition, which the authors proved to be correct using the CISE SMT solver [7].

2.3 State-Based Update Detection

State-based update detection means that operations are computed by comparing the persisted and current state of the tree-shaped data structure. The operations depend on the data model and there might be slight differences between the detected operations and those defined in the file system model. For $\mathcal{F}$ we detect:

- *createdir(i, p, n)*: a directory was created, when we find i with $type(i) = dir$ in the current state, but not in the persisted one
- *createfile′(i, p, n, c)*: a file with content c was created, when we find i with $type(i) = file$ in the current state, but not in the persisted one
- *move(i,u,v,n)*: an object was moved, when we find i in both states, but with varying name or parent
- *edit′(i)*: a file content was edited, when we find i in both states, but with different lastmodified meta-datum l. For update-detection, the exact content, i.e., *how* the file changed, is not relevant yet ($edit' \neq edit$)
- *delete′(i,p)*: an object was deleted when we find i in the persisted state, but not in the current one. $delete'$ is a *recursive* operation when it affects a *directory*. It aggregates all other detected $deletefile(j, q)$ and $deletedir(j, q)$ operations that affect objects j situated below i, i.e., where $ancestor(i, j)$ holds. When the synchronizer applies $delete'(i, p)$ to the other replica in the propagation stage, it has to apply the corresponding *deletefile* and *deletedir* operations according to a *post-order* traversal of the file system arborescence.

The computed list of operations does not indicate the exact order of operations, and some operations are affected by consolidation. See [4, 20] who identified this problem for file systems without *move* operation support. For $\mathcal{F}$ we find seven consolidation rules presented in Table 2 by examining all operation pairs. Note that $create = createfile \vee createdir$, $delete = deletefile \vee deletedir$.

Table 1. File system operations

Operation	Description, pre- and post-conditions
$\mathit{createdir}(i,p,n)$	Creates new dir with ID i and name n in parent dir with ID p Precondition: $\neg \mathit{ancestor}(i_{root},i) \wedge (\mathit{ancestor}(i_{root},p) \vee p = i_{root}) \wedge \mathit{type}(p) = \mathit{dir} \wedge \mathit{id}(p,n) = \mathit{error}$ Postcondition: $i \in \mathit{list}(p) \wedge \mathit{type}(i) = \mathit{dir} \wedge \mathit{lastmodified}(i) \neq \mathit{error}$
$\mathit{createfile}(i,p,n)$	Creates new file with ID i and name n in parent dir with ID p Precondition: see $\mathit{createdir}(i,p,n)$ Postcondition: $i \in \mathit{list}(p) \wedge \mathit{type}(i) = \mathit{file} \wedge \mathit{lastmodified}(i) \neq \mathit{error}$
$\mathit{move}(i,u,v,n)$	Moves a file or dir with ID i from parent dir with ID u to parent dir with ID v, and/or change the object's name to n Precondition: $\mathit{type}(u) = \mathit{dir} \wedge i \in \mathit{list}(u) \wedge \mathit{type}(v) = \mathit{dir} \wedge \mathit{id}(v,n) = \mathit{error} \wedge \neg \mathit{ancestor}(i,v)$ Postcondition: $i \in \mathit{list}(v) \wedge i \notin \mathit{list}(u)$ Note: $\neg \mathit{ancestor}(i,v)$ ensures that the user cannot move a dir to a destination dir below it.
$\mathit{deletefile}(i,p)$	Removes the file with ID i from parent dir with ID p Precondition: $\mathit{ancestor}(i_{root},i) \wedge \mathit{type}(i) = \mathit{file}$ Postcondition: $i \notin \mathit{list}(p) \wedge \neg \mathit{ancestor}(i_{root},i) \wedge \mathit{lastmodified}(i) = \mathit{error}$
$\mathit{deletedir}(i,p)$	Removes the empty dir with ID i from parent dir with ID p Precondition: $\mathit{ancestor}(i_{root},i) \wedge \mathit{type}(i) = \mathit{dir} \wedge \mathit{list}(i) = \{\}$ Postcondition: see $\mathit{deletefile}(i,p)$
$\mathit{edit}(i,\mathit{op})$	Changes the byte content of file with ID i by performing the operation op (e.g. adding, removing or changing bytes at specific positions within the file) Precondition: $\mathit{ancestor}(i_{root},i) \wedge \mathit{type}(i) = \mathit{file}$. Let $l_{pre} = \mathit{lastmodified}(i)$ Postcondition: $\mathit{ancestor}(i_{root},i) \wedge \mathit{lastmodified}(i) \neq l_{pre}$

3 Approach

This section describes our approach in generic steps, independent of a concrete data model, such as $\mathcal{F}$. It consists of two phases. The preparation phase described in Sect. 3.1 is done offline before implementing the software, whereas the execution phase applies the findings of phase 1, online at run-time of the synchronizer, see Sect. 3.2.

3.1 Phase 1: Preparation

In the preparation phase we get an understanding of the problems that can occur during synchronization by building and closely examining the file system model. We found that an analysis of the operation preconditions reveals two classes of issues: first, two concurrent operations (each applied to a different replica) can

Table 2. Operation consolidation rules

Operation consolidation rule	Explanation
$move(i, u, v_1, n_1) + move(i, v_1, v_2, n_2) \cong move(i, u, v_2, n_2)$	An object moved several times is detected as moved exactly once
$createfile(i, p, n) + edit(i, op) \cong createfile'(i, p, n, c)$	Creating an empty file and changing its content is detected as a non-empty file
$create(i, p, n_1) + move(i, p, v, n_2) \cong create(i, v, n_2)$	Creating and moving an object is detected as if it were created in the move operation's destination
$edit(i, op_1) + edit(i, op_2) \cong edit'(i)$	Editing a file multiple times is detected as a single *edit′* operation
$create(i, p, n) + delete(i, p) \cong []$	A created object that is subsequently deleted is not detected at all
$edit(i, op) + deletefile(i, p) \cong deletefile(i, p)$	When an edited file is subsequently deleted, only the deletion is detected
$move(i, u, v, n) + delete(i, v) \cong delete'(i, u)$	When a moved object is subsequently deleted, only the deletion is detected

cause *conflicts* that a synchronizer needs to handle. Second, state-based update detection will not detect the actual operations (and their order) applied by the user, but only an equivalent, unordered set. A precondition analysis must extract order dependencies (and even identify cycles), otherwise the synchronization of operations may fail. The following sections describe the individual steps.

Step 1: File System Model Formalization: The first step is to formally define the file system model the synchronizer uses internally, that consists of a formal definition of its elements, invariants and operations (with their pre- and postconditions). We recommend an automated approach where a model (initially built by hand) is iteratively refined via model checking tools, until all invariants and operations are known and free of contradictions. See [14] for an example, who did this for a model equivalent to our $\mathcal{F}$ model.

Step 2: Analysis of Conflicting Operations: An operation o_X detected in replica X is conflicting with operation o_Y detected in replica Y (and thus cannot be applied to Y by the synchronizer) if the preconditions of o_X no longer hold for new state of Y due to the modifications already applied to Y by o_Y.

To find conflicts, let OT be the list of *all* operation types of the model found in step 1. We start from an initially equal state for replicas X and Y. For any two types t_A, t_B from OT we instantiate operations o_X (of type t_A) and o_Y (of type t_B), apply o_X to X (which yields X') and o_Y to Y (yields Y'). We choose the operation parameters (e.g. i, p, u, v, n for $\mathcal{F}$) such that either applying o_Y to X', or o_X to Y' fails, due to violated preconditions.

Finding conflicts can be done manually or in an automated approach. The manual, pragmatic approach examines each individual precondition of each operation type t_A and finds a t_B, o_Y and o_X that produces a conflict. We generally recommend to identify *pseudo* conflicts, where two operations do conflict syntactically, but should not, because both operations have the same effect. In this case the synchronizer does not need to change the replicas, because the effect of both operations is the same anyway. An example for a pseudo-conflict is if o_X, o_Y are both *deletefile(i,p)* operations that affect the same object i.

Step 3: Resolving Conflicts: The general rule of conflict resolution is that the effect of operations o_X and o_Y are preserved as much as possible. There are two general approaches to conflict resolution:

1. Choose one of the operations to *win*, and manipulate the loser operation to resolve the conflict, or
2. Let both operations lose, by manipulating both of them, which avoids having to choose a winner.

We prefer option 1. Even though it is challenging to decide which of the two operations should take precedence in case of *automatic* resolution[2], the advantage is that at least one operation is fully preserved, and only the user who executed the loser operation needs to be informed. Our general approach for resolving conflicts is to perform the simplest possible resolution step, focusing on manipulating the *loser* operation instead of the winner operation. Sometimes the loser operation only has to be changed slightly, in other cases it has to be *undone* completely. Consider an example where o_X deletes a *directory* which was renamed by o_Y, and the strategy is to prefer *delete* over *move* operations. Instead of executing o_X in replica Y, which could cause side effects because the directory may have child-objects that are involved in other conflicts, it is more appropriate to undo o_Y. The winner operation o_X remains and is eventually executed, once all other conflicts have been resolved.

If conflict resolution is *automatic*, we need to make sure that if the preconfigured resolution was inappropriate for the user, the costs of subsequent, manual repair of the file system is manageable. Optimally, automatic resolutions can be changed by the user by a simple click, either before (via configuration) or after the fact.

In this step the synchronizer developer needs to examine each conflict found in step 2 and determine suitable resolution options. The operation(s) the synchronizer generates to resolve a conflict must be designed such that their execution cannot fail (due to violated preconditions), even if other conflicts exist.

[2] This is not a problem if the conflict resolution is delegated to the user.

Step 4: Analysis of Operation Order Dependencies: Assume that a file synchronizer has resolved conflicting operations between X and Y, such that the update detection now results in one set of unordered operations per replica $\bar{O}_X$, $\bar{O}_Y$. To be able to propagate the operations in $\bar{O}_X, \bar{O}_Y$, a suitable order needs to be found, which requires an analysis of the operation preconditions because not all operations are commutative. Let OT be the list of considered operation types. For any two types t_A, t_B from OT we instantiate the respective operations o_A, o_B, as they would have been *detected* (see Sect. 2.3) on one specific replica, e.g. X. We choose the parameters (e.g. i, p, u, v, n for $\mathcal{F}$) such that applying the sequence (o_A, o_B) to the other, unchanged replica is feasible, but applying (o_B, o_A) would fail, because a precondition of one of the two operations is violated . We end up with a list of *order dependencies*, where each order dependency contains t_A, t_B (in a specific order) and the violated operation precondition(s). Finally, we examine whether cycles can be built from the *order dependencies*.

3.2 Phase 2: Execution

Figure 1 provides a flow chart of our algorithm. Hexagons illustrate computation steps, table-shaped rectangles represent data structures. The *Current file system state* is provided (and regularly updated) by the update detection component of the synchronizer (not shown). Our algorithm is iterative. Let $\bar{O}_X, \bar{O}_Y$ be the detected operations. Step *Find conflicts* analyzes every operation pair of $\bar{O}_X, \bar{O}_Y$ and generates (1) a list of conflicts C where each $c \in C$ is a tuple of two conflicting operations, and (2) a list of *pseudo*-conflicts P, where each $p \in P$ summarizes two pseudo-conflicting operations. If $C \neq \emptyset$, C is sorted according to some preference (e.g. "resolve conflict type t_1 before type t_2"), if desired. Then a resolution operation is generated and executed that only resolves the first $c \in C$. If $C = \emptyset$ then operations are sorted according to Algorithm 1.

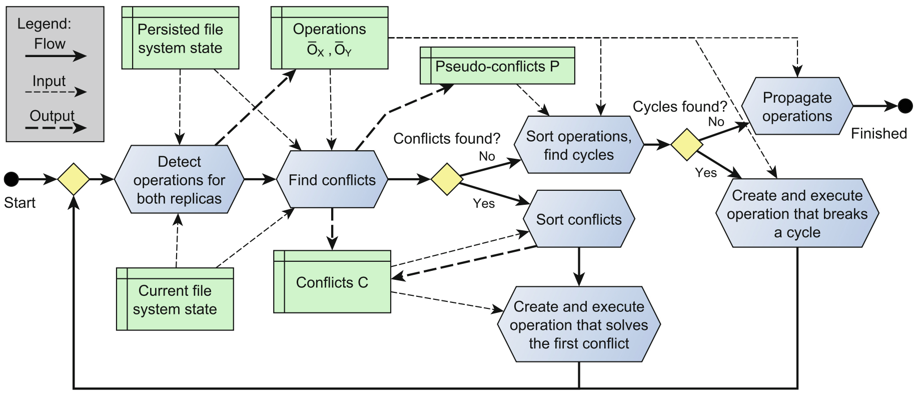
Legend:
Flow
Input
Output
Persisted file system state
Operations $\bar{O}_X, \bar{O}_Y$
Pseudo-conflicts P
Start
Detect operations for both replicas
Find conflicts
Conflicts found?
No
Yes
Sort operations, find cycles
Cycles found?
No
Yes
Propagate operations
Finished
Sort conflicts
Create and execute operation that breaks a cycle
Current file system state
Conflicts C
Create and execute operation that solves the first conflict

Fig. 1. Synchronization algorithm

Algorithm 1. Sorting operations

```
def sort_operations(Ox, Oy, P) -> L:
  global has_order_changed = False
  operations = [P + (Ox - P) + (Oy - P)]
  complete_cycles = []
  reorderings = []
  while True:
    has_order_changed = False
    find_and_fix_order_violations(operations)
    if not has_order_changed:
      break
    complete_cycles = find_complete_cycles(reorderings)
    if len(complete_cycles) > 0:
      break
  if len(complete_cycles) > 0:
    resolution_operation = break_cycle(complete_cycles[0])
    return [resolution_operation]
  else:
    return operations
```

We build `operations` as an initially unsorted list of pseudo-conflicting operations P and *non*-conflicting operations from $\bar{O}_X, \bar{O}_Y$ (that are not in P). Function `find_and_fix_order_violations()` performs an in-place sorting of `operations`. It checks all operation pairs for order violations as determined in step 4. If a violation is detected, the order of the two operations is swapped, the corrected order is added to `reorderings` and `has_order_changed` is set to *True*. Eventually either a cycle is found in `reorderings` which needs to be broken, or no more order violations were found in `operations`. In the first case `break_cycle()` must identify an operation o_X in the cycle where manipulating replica Y and the persisted state will dissolve a specific order dependency that involves o_X, turning the cycle into a chain. See Sect. 4.4 for an example. In the latter case our algorithm achieves convergence for both replicas, by iterating over each o in `operations` and executing the detected operation on the corresponding other replica, followed by updating the persisted state. If $o \in P$ then only the persisted state is updated, because the effect of o is already reflected in X and Y.

4 Application

In this section we provide an exemplary application of the four preparation steps from Sect. 3 to file system model $\mathcal{F}$.

4.1 Step 1: File System Model Formalization

Refer to the definition of $\mathcal{F}$ presented earlier in this work in Sect. 2.2.

4.2 Step 2: Conflict Detection

By examining the preconditions of the operations from Table 1, we find the conflicts and pseudo-conflicts presented in the following two lists. We use the $\otimes$ symbol for two conflicting operations. We use subscript letters X and Y as placeholders that designate to which replica the operation (or parameter) applies.

- **Create-Create**: On both replicas a new object was created with the same name under the same parent directory.
  Definition: $create_X(i_X, u_X, n_X) \otimes create_Y(i_Y, u_Y, n_Y) = [u_X = u_Y] \wedge [n_X = n_Y] \wedge [type_X(i_X) = dir \vee type_Y(i_Y) = dir \vee content_X(i_X) \neq content_Y(i_Y)]$ with $create := createdir \vee createfile'$.
  Violated precondition: $id(p, n) = error$
- **Edit-Edit**: The content of a file was changed on both replicas.
  Definition: $edit'_X(i_X, op_X) \otimes edit'_Y(i_Y, op_Y) = [i_X = i_Y] \wedge [content_X(i_X) \neq content_Y(i_Y)]$
  Violated precondition: technically no precondition is violated, but overwriting the file content on replica X with the one from replica Y would cause X's changes to be lost
- **Move-Create**: On one replica the user moved an object into a specific parent directory, assigning name n, on the other replica the user created a new object with the same name n in the same parent directory.
  Definition: $create_X(i_X, u_X, n_X) \otimes move_Y(i_Y, u_Y, v_Y, n_Y) = [u_X = v_Y] \wedge [n_X = n_Y]$ with $create := createdir \vee createfile'$
  Violated preconditions: create: $id(p, n) = error$; move: $id(v, n) = error$
- **Edit-Delete**: On one replica a file's content was edited, on the other replica the corresponding file was deleted.
  Definition: $edit'_X(i_X, op_X) \otimes delete'_Y(i_Y, p_Y) = (i_X = i_Y)$
  Violated precondition: On replica Y, $ancestor(i_{root}, i)$ of the $edit'$ operation is violated. On replica X there is no violation on the technical level, but on the semantic level: the changes of the $edit'$ operation would be lost.
- **Move-Delete**: On one replica an object was moved, on the other replica the corresponding object was deleted (either directly or as a consequence of deleting a parent directory).
  Definition: $move_X(i_X, u_X, v_X, n_X) \otimes delete'_Y(i_Y, p_Y) = (i_X = i_Y)$
  Violated precondition: On replica Y, $i \in list(u)$ of the *move* operation is violated. On replica X there is no violation on the technical level, but on the semantic level: the changes of the structural change of the move operation would be lost. The user who deleted the object would have done so without knowing that it was recently moved by another user on the other replica.
- **Move-Move (Source)**: On both replicas the same object was moved to a *different* location. That is, on each replica either the new name or parent directory (or both) differs.
  Definition: $move_X(i_X, u_X, v_X, n_X) \otimes move_Y(i_Y, u_Y, v_Y, n_Y) = (i_X = i_Y) \wedge [(v_X \neq v_Y) \vee (n_X \neq n_Y)]$
  Violated precondition: on replica X: $i_Y \in list(u_Y)$; on replica Y: $i_X \in list(u_X)$. The source is no longer in the expected location

- **Move-Move (Dest)**: The users of both replicas each moved a *different* object into the same parent directory, assigning the same name. The name of this conflict is Move-Move *(Dest)* because the conflict occurs at the *destination.*
  Definition: $move_X(i_X, u_X, v_X, n_X) \otimes move_Y(i_Y, u_Y, v_Y, n_Y) = (i_X \neq i_Y) \wedge (v_X = v_Y) \wedge (n_X = n_Y)$
  Violated precondition: $id(v, n) = error$
- **Move-ParentDelete**: On one replica the user deleted directory d, on the other replica the user moved another object into d.
  Definition: $move_X(i_X, u_X, v_X, n_X) \otimes delete'_Y(i_Y, p_Y) = (v_X = i_Y)$ $\wedge ancestor_Y(i_{root}, i_X)$
  Violated precondition: move: $type(v) = dir$
- **Create-ParentDelete**: On one replica the user deleted directory d, on the other replica the user creates a new object in d.
  Definition: $create_X(i_X, p_X, n_X) \otimes delete'_Y(i_Y, p_Y) = (p_X = i_Y)$ with $create := createdir \vee createfile'$
  Violated precondition: $type(p) = dir$
- **Move-Move (Cycle)**: Given two synchronized directories A and B, A was moved into B's namespace on one replica while B was moved into A's namespace on the other replica. This would create a cyclic parent-child relationship in the merged result.
  Definition: $move_X(i_X, u_X, v_X, n_X) \otimes move_Y(i_Y, u_Y, v_Y, n_Y) = (type(i_X) = type(i_Y) = dir) \wedge [ancestor(i_Y, v_X) \vee (i_Y = v_X)] \wedge [ancestor(i_X, v_Y) \vee (i_X = v_Y)]$ where *ancestor* refers to the state after all operations were executed.
  Violated precondition: $\neg ancestor(i, v)$

Pseudo-conflicts are presented in the following list, where $\odot$ indicates that two operations are pseudo-conflicting:

- **Edit-Edit**: The content of a file was changed on both replicas, such that the content is now the same.
- **Create-Create**: On both replicas a new *file* was created with the same content and name under the same parent directory. It would also be possible to consider two *create<u>dir</u>* operations to pseudo-conflict and to merge the directory contents recursively. However, if this resolution is done automatically and is inappropriate, manual clean up work is extensive [15].
- **Delete-Delete**: both replicas deleted the same object.
  Definition: $delete'_X(i_X, p_X) \odot delete'_Y(i_Y, p_Y) = (i_X = i_Y)$
- **Move-Move**: A specific object was moved to the same location.
  Definition: $move_X(i_X, u_X, v_X, n_X) \odot move_Y(i_Y, u_Y, v_Y, n_Y) = (i_X = i_Y) \wedge [(v_X = v_Y) \wedge (n_X = n_Y)]$

4.3 Step 3: Resolving conflicts

Who Wins? The winner of a conflict can be chosen in numerous ways. Either the user is explicitly involved in each decision, or conflicts are resolved automatically. For the latter the resolution strategy is pre-configured, typically by the developer. To better customize the synchronizer to the user's workflows, we propose to develop *multiple* conflict resolution strategies to choose from, where the choice may be given to the users or technically-apt administrators.

Name Occupation Conflicts: The **Create-Create**, **Move-Create** and **Move-Move (Dest)** conflicts all have in common that a specific name in a specific directory is being occupied by a *create* or *move* operation in each replica. A simple resolution approach is to modify the loser operation, by renaming the object on the corresponding replica, appending a unique suffix to the name.

Edit-Edit: When a specific file is edited on both replicas, undoing or modifying may not be possible because a replica may not store previous versions of a file. Resolving this conflict can either be achieved by renaming the loser file (and synchronizing it to the other replica, or keep it only on the loser replica), or backing up the loser file and overwriting it with the file of the winner replica, together with updating the *lastmodified* timestamp in the persisted state.

Delete Conflicts: Both **Edit-Delete** and **Move-Delete** are conflicts where one operation changed the object, while the other one deleted it. Thus the resolution approach should be similar for both. When the resolution favors the *delete* operation, a **Move-Delete** conflict can be resolved by undoing the move, but since the *edit* operation of an **Edit-Delete** conflict cannot be undone, the only solution is to delete the file from the loser replica and persisted state, to avoid the redetection of the conflict.

When the resolution favors the *move* or *edit* operation, we suggest the following approach:

- **Edit-Delete**: if the loser replica keeps deleted files in a garbage directory then restoring such files effectively undoes the *delete* operation. Otherwise the synchronizer can remove the file's entry from the persisted state only. In the next iteration, the file will be detected as *createfile′* operation and it will be synchronized to the loser replica.
- **Move-Delete**: the resolution works like for **Edit-Delete** conflicts. One caveat to consider is that when the *move* operation affects a *directory*, removing its entries from the database may cause orphaned entries for those child-objects that were moved out. For instance, given a directory at path '/d' and file '/d/f', with operations $move_X(\text{'}/d\text{'}, \text{'}/e\text{'}), move_Y(\text{'}/d/f\text{'}, \text{'}/f\text{'}), delete'_Y(\text{'}/d\text{'})$. To restore the directory, deleting *both* the directory and its children from the persisted state is inappropriate, because then one *move* operation would be lost, causing file f to be duplicated. However, removing only those objects that were deleted on replica Y, here: '/d', would also be inappropriate, because f would then be orphaned in the persisted state. We propose to move such orphaned objects temporarily to the root level in the persisted state and solve follow-up **Move-Move (Source)** conflicts in favor of replica Y.

Move-Move (Source): We propose to resolve this conflict type by undoing the loser *move* operation. Note that undoing a *move* operation may not always be possible: the source parent directory s might already be deleted, or the original name of the moved object might already be occupied in s, or the user could have moved s such that it is now a child of the affected object. In case of such issues we propose to move the affected object to the root of the synchronized directory instead, with a random suffix added to its name.

Indirect Conflicts: Two operations indirectly conflict with each other if they don't target two *different* objects, which are always in a hierarchical parent-child relationship. The **Move-Move (Cycle)**, **Move-ParentDelete** and **Create-ParentDelete** conflict belong to this category. **Move-Move (Cycle)** conflicts can be resolved exactly like **Move-Move (Source)** conflicts. **Move-ParentDelete** can be resolved by either undoing the deletion by restoring the deleted directory *in its entirety* (with all sub-objects), if possible, or to prefer the *delete* operation by undoing the *move* operation. The goal is to resolve this conflict in a way that avoids that *both* users are unhappy with the merged result. For instance, the two following resolution approaches would be bad ideas: (1) favor the *move* operation, by restoring only the deleted directory (and all its ancestor directories) targeted by the *move* operation, in order to make the *move* operation possible. This would *partially* undo the *delete* operation and cause an inconsistent namespace that would not be appreciated by either user. (2) Favor the *delete* operation by deleting the directory. This would cause the moved file to be deleted, which was not the intention of either user. In contrast, our solution only discards the intention for the user of the *move* operation.

Resolving **Create-ParentDelete** conflicts works similarly. We also suggest to take precedence to the *delete* operation. Undoing the *create* operation would mean data loss, thus we suggest to back up the created object first, or to move it to the root of the synchronized directory, or to a garbage directory.

Pseudo Conflicts: A pseudo conflict is resolved by updating the entries of the affected objects in the persisted state, such that the two operations are no longer detected in the next iteration. For example, a *Delete-Delete* pseudo-conflict would be resolved by removing the entries of the affected objects from the persisted state.

4.4 Step 4: Analysis of Operation Order Dependencies

For $\mathcal{F}$ we choose $OT = \{createfile', createdir, move, edit', delete'\}$. Figure 2 shows an overview of the eight order dependencies we found for the operation types in OT. The arrows are denoted with a dependency number explained below:

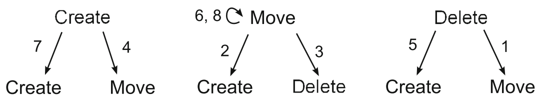

Fig. 2. Operation dependencies

1. *delete′* before *move*, e.g. user deletes an object at path '/x' and moves another object '/a' to '/x'
2. *move* before *create*, e.g. user moves an object '/a' to '/b' and creates another object at '/a'
3. *move* before *delete′*, e.g. user moves object '/X/y' outside of directory '/X' (e.g. to '/z') and then deletes '/X'
4. *create* before *move*, e.g. user creates directory '/X' and moves object '/y' into '/X'
5. *delete′* before *create*, e.g. user deletes object '/x' and then creates a new object at '/x'
6. *move* before *move* (occupation), e.g. user moves file '/a' to '/temp' and then moves file '/b' to '/a'
7. *create* before *create*, e.g. user creates directory '/X' and then creates an object inside it
8. *move* before *move* (parent-child flip), e.g. user moves directory '/A/B' to '/C', then moves directory '/A' to '/C/A' (parent-child relationships are now flipped)

By connecting the dependencies, we're able to construct *cycles*. Figure 3 shows *minimal* cycles (with the smallest possible number of operations) in the first row, and two examples of more elaborate cycles in the second row. We note that it is impossible to build cycles of only *delete* and *create* operations. It is also easy to prove that cycles that exclusively consist of *move* operations connected only by rule *8* are impossible[3]. Cycles always include at least one *move* operation.

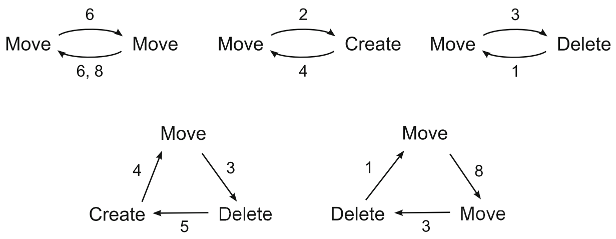

Fig. 3. Operation dependency cycles

[3] Intuitively, a proof by contradiction shows that the existence of a *rule 8 cycle* of n objects would require that those n objects also formed a cyclic parent-child relationship (before and after synchronization), but cycles are not allowed in $\mathcal{F}$.

For any cycle found in replica X there must always be at least one operation o_X (that affects object with ID i) which frees a location (i.e., a name in a specific directory) that is used by a follow-up operation o'_X. o_X must either be a *move* (dependencies $6 + 2$) or a *delete* (dependencies $1 + 5$) operation. Instead of executing o_X we generate a different *move* operation r_Y that breaks the cycle. r_Y renames i by appending a unique suffix to its name. We execute r_Y on Y and the persisted state. This way, r_Y is not detected after restarting the algorithm, but o_X still is detected because r_Y did not modify X: if o_X is a *move* operation, changing the name of i in the persisted state to a *unique* name will still find i as moved on X; if o_X is a *delete* operation then it will still be deleted on X. However, the cycle is now broken, because the order dependency (6, 2, 1, or 5) no longer applies. Note that if $o_X \in P$, i.e., o_X is a pseudo-conflicting operation, r_Y may only be executed on the persisted state, leaving both physical replicas X, Y untouched.

5 Evaluation

We implemented the approach presented in Sect. 4 as a user-space Python program that synchronizes folders on the user's local disk to the BSCW groupware [16]. We deployed it to 30 users who have been using it in production for over one year. In addition to hundreds of hand-made tests we applied two automated testing approaches to verify practical correctness of our algorithm. We used a variation of model checking.

In the first test approach we generated all possible operation sequences that can be applied to the 12 start scenarios that consist of three directories and one file. Due to state-space explosion we limited the number of operations to one *createfile*, two *createdir*, three *move* and three *delete* operations. This resulted in 5.5 million test cases computed in a HPC cluster over several weeks. Because local file systems (even RAM disks) are slow, we sped up test generation and execution by implementing a simple in-memory file system used instead.

To overcome the operation count limit resulting from state explosion in the first test approach, the second approach generated a much larger count (up to 30 operations). Each operation type and its parameters were chosen at random. We ran millions of test cases and discovered no anomalies in our implementation.

6 Conclusions

In this work we presented an iterative algorithm for the synchronization of two replicas X and Y that hold tree-shaped data structures, where operations since the last synchronization are detected using a *state-based* approach. We applied it to *file synchronizers* with a concrete file system model.

While the drawback of our iterative algorithm is the increased run-time in those scenarios where multiple iterations are required, we observed that those higher costs only occur after long offline periods. The advantage of the algorithm is that its individual steps are simple to implement, minimal and atomic.

Therefore the synchronization procedure can be interrupted any time, because it avoids long-lasting transactions.

This work demonstrates two challenges during synchronization. First, state-based update detection does not provide the order of the detected operations, which we solve by analyzing the operation preconditions to find a suitable order. The second challenge is that an operation in replica X may conflict with another operation in Y. We provide guidelines for how to identify sensible resolution options, find all possible conflicts, and how to build operations that resolve them. We leave finding a suitable (graphical) representation of the conflicts and their resolution (if automatic) as future work. We consider such conflict awareness an important aspect, as it improves the overall usability of the system.

References

1. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 3–46. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88643-3_1
2. 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, MobiCom 1998, pp. 98–108. ACM, New York (1998). https://doi.org/10.1145/288235.288261
3. Bao, X., Xiao, N., Shi, W., Liu, F., Mao, H., Zhang, H. (eds.): SyncViews: toward consistent user views in cloud-based file synchronization services. In: 2011 Sixth Annual Chinagrid Conference (2011). https://doi.org/10.1109/ChinaGrid.2011.35
4. Csirmaz, E.: Algebraic File Synchronization: Adequacy and Completeness (2016). https://arxiv.org/pdf/1601.01736.pdf
5. Dearman, D., Pierce, J.S.: It's on my other computer! Computing with multiple devices. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 767–776. ACM, Florence (2008). https://doi.org/10.1145/1357054.1357177
6. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Exploiting schemas in data synchronization. J. Comput. Syst. Sci. **73**(4), 669–689 (2007). https://doi.org/10.1016/j.jcss.2006.10.024
7. Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: Cause I'm strong enough: reasoning about consistency choices in distributed systems. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 371–384. ACM, St. Petersburg (2016). https://doi.org/10.1145/2837614.2837625
8. Jokela, T., Ojala, J., Olsson, T.: A diary study on combining multiple information devices in everyday activities and tasks. In: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, pp. 3903–3912. ACM, Seoul (2015). https://doi.org/10.1145/2702123.2702211
9. Kollmar, F.: The Cloud Storage Report - Dropbox Owns Cloud Storage on Mobile (2016). https://blog.cloudrail.com/cloud-storage-report-dropbox-owns-cloud-storage-mobile/
10. Li, Q., Zhu, L., Zeng, S., Shang, W.Q. (eds.): An improved file system synchronous algorithm. In: 2012 Eighth International Conference on Computational Intelligence and Security (2012). https://doi.org/10.1109/CIS.2012.123

11. Li, Q., Zhu, L., Shang, W., Zeng, S.: CloudSync: multi-nodes directory synchronization. In: 2012 International Conference on Industrial Control and Electronics Engineering (ICICEE 2012), Piscataway, NJ, pp. 1470–1473. IEEE (2012). https://doi.org/10.1109/ICICEE.2012.386
12. Lindholm, T., Kangasharju, J., Tarkoma, S.: A hybrid approach to optimistic file system directory tree synchronization. In: Kumar, V., Zaslavsky, A., Cetintemel, U., Labrinidis, A. (eds.) The 4th ACM International Workshop on Data Engineering for Wireless and Mobile Access, pp. 49–56. ACM, New York (2005). https://doi.org/10.1145/1065870.1065879
13. Molli, P., Oster, G., Skaf-Molli, H., Imine, A.: Using the transformational approach to build a safe and generic data synchronizer. In: Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work, pp. 212–220. ACM, Sanibel Island (2003). https://doi.org/10.1145/958160.958194
14. Najafzadeh, M.: The analysis and co-design of weakly-consistent applications. Ph.D. thesis, Université Pierre et Marie Curie (2016). https://hal.inria.fr/tel-01351187/document
15. 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, pp. 61–70. ACM, Sanibel Island (2016). https://doi.org/10.1145/2957276.2957278
16. OrbiTeam Software GmbH & Co KG: BSCW Social (2018). https://www.bscw.de/social/
17. Pierce, B.C., Vouillon, J.: What's in unison? A formal specification and reference implementation of a file synchronizer (2004)
18. Price, R.: Google Drive now hosts more than 2 trillion files (2017). http://www.businessinsider.de/2-trillion-files-google-drive-exec-prabhakar-raghavan-2017-5
19. Qian, Y.: Data synchronization and browsing for home environments. Ph.D. thesis, Eindhoven University of Technology (2004)
20. Ramsey, N., Csirmaz, E.: An algebraic approach to file synchronization. In: Tjoa, A.M., Gruhn, V. (eds.) the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium, p. 175 (2001). https://doi.org/10.1145/503209.503233
21. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. **37**(1), 42–81 (2005). https://doi.org/10.1145/1057977.1057980
22. Santosa, S., Wigdor, D.: A field study of multi-device workflows in distributed workspaces. In: Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing, pp. 63–72. ACM, Zurich (2013). https://doi.org/10.1145/2493432.2493476
23. Tao, V., Shapiro, M., Rancurel, V.: Merging semantics for conflict updates in geo-distributed file systems. In: Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, pp. 10:1–10:12. ACM, New York (2015). https://doi.org/10.1145/2757667.2757683
24. Torvalds, L., Hamano, J.: Git: Distributed Version Control (2010). https://git-scm.com
25. Vonrueden, M., Prinz, W.: Distributed document contexts in cooperation systems. In: Kokinov, B., Richardson, D.C., Roth-Berghofer, T.R., Vieu, L. (eds.) CONTEXT 2007. LNCS (LNAI), vol. 4635, pp. 507–516. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74255-5_38