Theory Of File Synchronization

Előd P. Csirmaz

under the direction of
Dr. Norman Ramsey
Harvard University

revised version

Research Science Institute
July 29, 2000

Abstract

The problem of file synchronization (making two, differently modified copies of a filesystem the same again without losing information) emerges in many cases, and is solved in many different ways. Our goal was to create a *general mathematical model* for file synchronization and use it as a basis to define the behavior of a synchronizer.

We developed a specific algebraic model on filesystem commands and proved that it is sound and complete for its intended interpretation on real systems, that is, if commands are considered to be equivalent according to the algebra then they are equivalent when applied to a real filesystem and vice versa.

Then we defined algorithms for synchronization using our algebra and created an implementation which was tested on various filesystems. This method turned out to be an effective way to create the specification of a synchronizer since it simplified both the definition and the implementation.

The methods used in the proofs for soundness and completeness are likely be usable on other algebras, too. Thus using algebraic approach could make it possible to extend the synchronizer to other types of datasystems and to create a general theory of synchronization in the future.

Contents

1 Introduction

If we have multiple copies (called *replicas*) of a filesystem or a part of a filesystem (for example on a laptop and on a desktop computer), it is often the case that changes are made to each replica independently and as a result they do not contain the same information. In that case, a *file synchronizer* is used to make them similar (consistent) again, without losing any information.

The goal of a file synchronizer is to *detect conflicting updates* and *propagate non–conflicting updates on each replica.* Due to the various cases of file updates and the various behaviors of filesystem commands it is difficult to give a complete and correct definition of its behavior.

It can be helpful in solving this problem if we have a formalized mathematical model for describing file synchronization.

We investigated algebraic models of synchronization in a specific case to see the properties of such a system and the method of proving theorems in it which will hopefully aid us in creating a general theory of synchronization based on algebraic structure in the future. This specific case was the synchronization of filesystems.

In the paper we develop an algebraic representation of filesystem commands. Then we show that this model is both complete and sound, and after that we define the expected behavior of a file synchronizer and create its specification with the help of our algebra.

This method will hopefully simplify the specification as well as the implementation of the synchronizer, and it might make it possible to extend the synchronizer to other types of datasystems, such as mail folders or databases.

1.1 The main idea of synchronization

In general, there are two phases of the task of a synchronizer: *update detection*, when the program recognizes where updates have been made since the last synchronization and *reconciliation*, when it combines updates to yield a new, synchronized version of each replica. If there are no conflicting updates, then at the end of the synchronization the replicas will be the same. Otherwise a synchronizer may leave them unchanged at paths where conflicts occurred and warn the user.

In this paper, we focus on the commands that were applied to these replicas while they were modified independently. The main aim of a synchronizer is to perform *all* commands on *all* replicas.

Let us introduce some notation. We have the original system, O; and the present replicas $R_1, R_2, \ldots R_n$. Somehow we know the sequences of commands which were applied to O in each replica (for example from the update detector). They are $S_1, S_2, \ldots S_n$. The synchronizer algorithm will provide us commands on each replica ($S_1^*, S_2^*, \cdots S_n^*$) which lead each replica to a common state O' if possible.

In order to determine each sequence S_i^*, we take all commands applied to the systems ($S_1 \cup S_2 \cup \cdots \cup S_n$) then omit commands in S_i which were already applied to replica R_i.

But that way we only gain a *set* of commands without order. We must order it somehow to be able to apply the commands to the system. For it is possible that some orders cause errors in a filesystem (for example, trying to remove a directory before removing the files in it). It is also possible that not all error–free orders have the same effect on the filesystem. For example, modifying the contents of a file to "xxx" and modifying them to "yyy" are not commutable commands: they leave the system in different states if applied in different order. In this case the synchronizer cannot synchronize the systems fully, since "last modifying wins" is not a good solution in every case. Therefore the aim of the synchronizer is to find commands for which *all error–free orders have the same effect* and apply them to the system.

The reason we may have two differently ordered sequences of commands which do not lead a system to the same state is that we have incommutable pairs of commands in them. This is because if all pairs of commands commuted (that is, pair of commands $C_1; C_2$ had the same effect on a system as $C_2; C_1$) then clearly one of the sequences could be changed to the other one by commuting commands without any change in the effect of the sequence on the system.

Because of this, we focused on the commutability of pairs of commands. We defined this property with the help of an algebraic proof system on commands.

2 Definitions

2.1 The filesystem

We introduce some known notation on filesystems and paths.

A filesystem can be *broken* ($F = \perp$) or it can be a function mapping whole paths to their contents. The filesystem is broken if a command caused an error, for example deleting a directory with files under it.

A *path* is either empty (/), or a finite sequence of names separated by /. The concatenation of paths π and φ can be written as π/φ. We write $\pi \preceq \gamma$ iff π is a prefix of γ, i.e., if $\gamma = \pi/\alpha$ for some path α which might be empty. We also write $\pi \prec \gamma$ if π is a proper prefix of γ, that is, $\pi \preceq \gamma$ and $\pi \neq \gamma$. If we refer to the contents of path π in filesystem F, we write simply $F(\pi)$.

In the paper, π_{sub} always refers to a file or directory under π, that is, $\pi \prec \pi_{sub}$. For $\pi_{\varepsilon sub}$, $\pi \preceq \pi_{\varepsilon sub}$ holds. The same for the opposite direction: $\pi^{sup} \prec \pi$ and $\pi^{\varepsilon sup} \preceq \pi$. Paths π and φ are usually incomparable paths.

In a filesystem, the contents of a path can be broken ($F(\pi) = \perp$, if the file or directory does not exist), they can be a file ($F(\pi) = X_{File}$) or a directory ($F(\pi) = Y_{Dir}$). We also write X or Y for some unspecified but non-$\perp$ contents. Both files and directories can carry additional information (modtimes, permissions, the bytes in a file, etc.), which we leave unspecified, simply writing the contents as shown above.

Filesystems are functions and are compared using extensional equality. Two filesystems F_1 and F_2 are equivalent iff $(F_1 = \perp) \wedge (F_2 = \perp)$, *or* $\forall \pi : F_1(\pi) = F_2(\pi)$.

We write $F\{\pi \mapsto X\}$ for the function that is like F, except it maps π to X. So

$$F\{\pi \mapsto X\}(\gamma) = \begin{cases} X, & \text{if } \pi = \gamma \\ F(\gamma), & \text{otherwise} \end{cases}$$

We write $\mathit{childless}_F(\pi)$ iff $F(\pi)$ has no descendants, i.e. $\forall \gamma : \pi \prec \gamma \implies F(\gamma) = \perp$.

$\mathit{parent}(\pi)$ denotes the path which immediately precedes π, that is, for some name $\mathbf{p}$, $\mathit{parent}(\pi)/\mathbf{p} = \pi$.

S always refers to sequences of commands. SF is the filesystem obtained by applying all commands in S to F. We may also write a command instead of S. $S_1; S_2$ refers to the concatenation of sequences S_1 and S_2.

We have some restrictions on filesystems: all filesystems must satisfy the *tree property*, that is, if $\pi \prec \gamma$ and $F(\gamma) \neq \perp$ then $F(\pi) = X_{Dir}$. It means that every path which has a descendant must be a directory.

2.2 Commands

Now we introduce the basis of our algebra: the commands on filesystems.

At first, we considered using four commands: *create*, *remove*, *edit* and *move*, as the most common commands on a filesystem. Later, *move* turned out to be hard to handle in the algebra. We decided to introduce this command only in the user interface, after running the synchronizer. We also needed a command to reason about commands that cause errors (*break*). So we took another set of commands; it consists of the commands *create*, *remove*, *edit* and *break*. In appendix C we provide an argument to justify using a move–free algebra.

All the commands have the following property: if applied to a filesystem, then it either breaks the system ($\mathit{command}(\pi, X)F = \perp$) or $\mathit{command}(\pi, X)F = F\{\pi \mapsto X\}$; that is, if a command does not break the filesystem, it only affects the filesystem at the path on which it was applied.

Now we define the exact behavior of each command. Applied to the broken filesystem, they all have the same effect: leaving the filesystem in the broken state. Otherwise, the definition of their effect is the following:

- $\mathit{create}(\pi, X)$ modifies $F(\pi)$ to X iff $F(\pi)$ is broken and its parent is a directory; i.e.

$$\mathit{create}(\pi, X)F = \begin{cases} F\{\pi \mapsto X\}, & \text{iff } F(\pi) = \perp \wedge F(\mathit{parent}(\pi)) = Y_{Dir} \\ \perp, & \text{otherwise.} \end{cases}$$

- The command *edit* can change the type of a path (file or directory), according to its second argument. Since the tree–property must be preserved, the definition is:

$$edit(\pi, X_{Dir})F = \begin{cases} F\{\pi \mapsto X_{Dir}\}, & \text{iff } F(\pi) \neq \bot \\ \bot & \text{otherwise;} \end{cases}$$

$$edit(\pi, X_{File})F = \begin{cases} F\{\pi \mapsto X_{File}\} & \text{iff } F(\pi) \neq \bot \wedge childless_F(\pi) \\ \bot & \text{otherwise.} \end{cases}$$

- $remove(\pi)$ only removes files or directories without files:

$$remove(\pi)F = \begin{cases} F\{\pi \mapsto \bot\} & \text{iff } F(\pi) \neq \bot \wedge childless_F(\pi) \\ \bot & \text{otherwise.} \end{cases}$$

- *break* breaks the filesystem:

$$breakF = \bot \quad \text{for every } F.$$

We write *skip* for the empty sequence of commands.

3 Algebra on commands

In order to reason about commands, we define *algebraic laws* on commands. We use formal logic to build a proof system which is sound and complete for its intended interpretation.

For sequences S_1 and S_2, we have two kinds of judgements:

- $S_1 \equiv S_2$, or S_1 *is algebraically equivalent to* S_2. Its intended interpretation is that they act the same on all filesystems, i.e. $\forall F : S_1F = S_2F$.
- $S_1 \sqsubseteq S_2$, or S_2 *extends* S_1; its intended interpretation is the following: if $S_1 \sqsubseteq S_2$ and $S_1F \neq \bot$ then $S_1F = S_2F$.

The axioms of our proof system are the laws listed in Table 1. We have two inference rules, one for each judgement.

3.1 The laws

We created the laws with the help of pairs of commands. Lines in the last section are not axioms; they are written there to list all possible pairs.

How many pairs do we have? From *edit*, *create* and *remove*, we can choose a pair of commands in 9 ways. Moreover, there are four types of path-pairs: $\pi;\pi$, $\pi_{sub};\pi$, $\pi;\pi_{sub}$ and $\pi;\varphi$. That makes 4×9 cases. With *break*, we have 3×2 cases with an other non–break command (there are 3 of them and they can follow or precede *break*) and one case when we have a pair made of two *break*s. Thus, we have $4 \times 9 + 3 \times 2 + 1 = 43$ pairs.

We subdivide some laws containing *edit* because of *edit*'s different behavior when modifying paths to files or to directories. These laws are numbered with A or B. Extension laws (that is, where the relation is $\sqsubseteq$, not $\equiv$) are marked with E.

3.2 Inference rules

An inference rule makes it possible to derive new statements from axioms or others statements known to be already true. The inference rules are:
For any sequences S_1, S_2, S, S'*:*

Commuting laws

1. $\mathit{edit}(\pi, X); \mathit{edit}(\pi_{sub}, Y) \equiv \mathit{edit}(\pi_{sub}, Y); \mathit{edit}(\pi, X)$
2. $\mathit{edit}(\pi_{sub}, Y); \mathit{edit}(\pi, X) \equiv \mathit{edit}(\pi, X); \mathit{edit}(\pi_{sub}, Y)$

4A$_E$. $\mathit{create}(\pi_{sub}, Y); \mathit{edit}(\pi, X_{Dir}) \sqsubseteq \mathit{edit}(\pi, X_{Dir}); \mathit{create}(\pi_{sub}, Y)$
5A. $\mathit{edit}(\pi, X_{Dir}); \mathit{remove}(\pi_{sub}) \equiv \mathit{remove}(\pi_{sub}); \mathit{edit}(\pi, X_{Dir})$
6A. $\mathit{remove}(\pi_{sub}); \mathit{edit}(\pi, X_{Dir}) \equiv \mathit{edit}(\pi, X_{Dir}); \mathit{remove}(\pi_{sub})$

7. $\mathit{edit}(\pi, X); \mathit{edit}(\varphi, Y) \equiv \mathit{edit}(\varphi, Y); \mathit{edit}(\pi, X)$
8. $\mathit{edit}(\pi, X); \mathit{create}(\varphi, Y) \equiv \mathit{create}(\varphi, Y); \mathit{edit}(\pi, X)$
9. $\mathit{edit}(\pi, X); \mathit{remove}(\varphi) \equiv \mathit{remove}(\varphi); \mathit{edit}(\pi, X)$
10. $\mathit{create}(\varphi, Y); \mathit{edit}(\pi, X) \equiv \mathit{edit}(\pi, X); \mathit{create}(\varphi, Y)$
11. $\mathit{create}(\pi, X); \mathit{create}(\varphi, Y) \equiv \mathit{create}(\varphi, Y); \mathit{create}(\pi, X)$
12. $\mathit{create}(\pi, X); \mathit{remove}(\varphi) \equiv \mathit{remove}(\varphi); \mathit{create}(\pi, X)$
13. $\mathit{remove}(\varphi); \mathit{edit}(\pi, X) \equiv \mathit{edit}(\pi, X); \mathit{remove}(\varphi)$
14. $\mathit{remove}(\varphi); \mathit{create}(\pi, X) \equiv \mathit{create}(\pi, X); \mathit{remove}(\varphi)$
15. $\mathit{remove}(\pi); \mathit{remove}(\varphi) \equiv \mathit{remove}(\varphi); \mathit{remove}(\pi)$

Breaking laws

3B. $\mathit{edit}(\pi, X_{File}); \mathit{create}(\pi_{sub}, Y) \equiv \mathit{break}$
4B. $\mathit{create}(\pi_{sub}, Y); \mathit{edit}(\pi, X_{File}) \equiv \mathit{break}$
5B. $\mathit{edit}(\pi, X_{File}); \mathit{remove}(\pi_{sub}) \equiv \mathit{break}$

16. $\mathit{edit}(\pi, X); \mathit{create}(\pi, Y) \equiv \mathit{break}$
17. $\mathit{edit}(\pi_{sub}, X); \mathit{create}(\pi, Y) \equiv \mathit{break}$
18. $\mathit{edit}(\pi_{sub}, X); \mathit{remove}(\pi) \equiv \mathit{break}$
19. $\mathit{create}(\pi, X); \mathit{edit}(\pi_{sub}, Y) \equiv \mathit{break}$
20. $\mathit{create}(\pi, X); \mathit{create}(\pi, Y) \equiv \mathit{break}$
21. $\mathit{create}(\pi_{sub}, X); \mathit{create}(\pi, Y) \equiv \mathit{break}$
22. $\mathit{create}(\pi, X); \mathit{remove}(\pi_{sub}) \equiv \mathit{break}$
23. $\mathit{create}(\pi_{sub}, X); \mathit{remove}(\pi) \equiv \mathit{break}$
24. $\mathit{remove}(\pi); \mathit{edit}(\pi, X) \equiv \mathit{break}$
25. $\mathit{remove}(\pi); \mathit{edit}(\pi_{sub}, X) \equiv \mathit{break}$
26. $\mathit{remove}(\pi); \mathit{create}(\pi_{sub}, X) \equiv \mathit{break}$
27. $\mathit{remove}(\pi_{sub}); \mathit{create}(\pi, X) \equiv \mathit{break}$
28. $\mathit{remove}(\pi); \mathit{remove}(\pi) \equiv \mathit{break}$
29. $\mathit{remove}(\pi); \mathit{remove}(\pi_{sub}) \equiv \mathit{break}$

Simplifying laws

30$_E$. $\mathit{edit}(\pi, X); \mathit{edit}(\pi, Y) \sqsubseteq \mathit{edit}(\pi, Y)$
31. $\mathit{edit}(\pi, X); \mathit{remove}(\pi) \equiv \mathit{remove}(\pi)$
32. $\mathit{create}(\pi, X); \mathit{edit}(\pi, Y) \equiv \mathit{create}(\pi, Y)$
33$_E$. $\mathit{create}(\pi, X); \mathit{remove}(\pi) \sqsubseteq \mathit{skip}$
34$_E$. $\mathit{remove}(\pi); \mathit{create}(\pi, X) \sqsubseteq \mathit{edit}(\pi, X)$

Laws for *break*

35. $\mathit{break}; \mathit{edit}(\pi, X) \equiv \mathit{break}$
36. $\mathit{break}; \mathit{create}(\pi, X) \equiv \mathit{break}$
37. $\mathit{break}; \mathit{remove}(\pi) \equiv \mathit{break}$
38. $\mathit{edit}(\pi, X); \mathit{break} \equiv \mathit{break}$
39. $\mathit{create}(\pi, X); \mathit{break} \equiv \mathit{break}$
40. $\mathit{remove}(\pi); \mathit{break} \equiv \mathit{break}$
41. $\mathit{break}; \mathit{break} \equiv \mathit{break}$

Remaining pairs
(no substitution)

3A. $\mathit{edit}(\pi, X_{Dir}); \mathit{create}(\pi_{sub}, Y) \sqsupseteq \mathit{create}(\pi_{sub}, Y); \mathit{edit}(\pi, X_{Dir})$
6B. $\mathit{remove}(\pi_{sub}); \mathit{edit}(\pi, X_{File})$
42. $\mathit{create}(\pi, X); \mathit{create}(\pi_{sub}, Y)$
43. $\mathit{remove}(\pi_{sub}); \mathit{remove}(\pi)$

Table 1: Algebraic laws

- if $S_1 \equiv S_2$ then $S; S_1; S' \equiv S; S_2; S'$
- if $S_1 \sqsubseteq S_2$ then $S; S_1; S' \sqsubseteq S; S_2; S'$
- if $S_1 \equiv S_2$ then $S_1 \sqsubseteq S_2$.

The first inference rules merely substitute part of a sequence for another sequence. If "$S_1 \equiv S_2$" or "$S_1 \sqsubseteq S_2$" is a law itself, we say that we applied the law to the sequence $S; S_1; S'$.

3.3 Soundness theorem

In order to be able to use our algebra, we must show that algebraic relation between sequences also means that the sequences act the same on filesystems. It can be proven that for every two sequences of commands

$$S \equiv S^* \implies \forall F : SF = S^*F,$$
$$(S \sqsubseteq S^*) \wedge (SF \neq \perp) \implies \forall F : SF = S^*F.$$

The proof for the two cases are very similar. Induction is used on the number of times inference rules are applied to the sequences. The soundness of each individual law (which can be done by investigating a number of cases) and the inference rules is shown separately.

For the detailed proof see Appendix A.

3.4 Theorem of completeness

In this section, we show that our proof system is complete for its interpretation. This is also mandatory to be able to use our algebra.

Let us introduce some notation. We write $S_1 \parallel S_2$, or S_1 *and* S_2 *have a common upper bound* iff $\exists S^* : S_1 \sqsubseteq S^* \wedge S_2 \sqsubseteq S^*$, that is, iff both S_1 and S_2 can be extended to the same sequence. It is a symmetric relation, but not transitive. $S_1 \equiv S_2 \implies S_1 \sqsubseteq S_2 \implies S_1 \parallel S_2$ also holds.

Now we prove that if two sequences of commands act the same on any filesystem neither of them breaks, *and* there is a filesystem neither of them breaks then they have a common upper bound. Formally,

$$\begin{aligned}
&\forall S, S' : \\
&\qquad ((\forall G : (SG \neq \bot \wedge S'G \neq \bot) \implies SG = S'G) \\
&\qquad \wedge \\
&\qquad (\exists F : SF \neq \bot \wedge S'F \neq \bot) \\
&\qquad \implies S \parallel S'),
\end{aligned}$$

where G and F refer to filesystems.

In the proof we define *minimal sequences* in the following way: consider the set of sequences $\wp_S = \{S^* | S \sqsubseteq S^*\}$. Because of our preconditions, the sequence *break* is not in $\wp_S$ (if it was, that is, $S \sqsubseteq \mathit{break}$, $SF = \bot$ would hold since by definition $SF = (\mathit{break})F$ if $SF \neq \bot$).

Let S_0 be (one of) the shortest sequence(s) in $\wp_S$ and, similarly, S'_0 (one of) the shortest sequence(s) in $\wp_{S'}$. These sequences are called minimal sequences. It can be shown that for every G which satisfies the condition $SG \neq \bot \wedge S'G \neq \bot$, $S_0G = S'_0G \neq \bot$ applies. As a special case, we know that $S_0F = S'_0F \neq \bot$.

The proof has three main steps.

i. We have some constraints on S_0 and S'_0. Since they do not break every filesystem, no breaking laws can be applied to them. And since they have minimal length, we cannot apply a simplifying law either. From these properties we can prove that there is at most one command on each path in a minimal sequence. (If there were more, such a law would be applicable.)

ii. We know that a command on path π only affects the filesystem at π if it does not break the filesystem. Therefore S_0 and S'_0 must contain commands on the same paths since they do not break F and $S_0F = S'_0F$, that is, they modify F at the same points. And we can also prove that the sequences must contain the same commands on each path from the fact that they act the same on every G. Therefore they consist of the very same commands.

iii. We prove that S_0 and S'_0 can be reordered by commuting laws so that they would be the same sequences. That is, a sequence S^* exists for which $S_0 \sqsubseteq S^* \sqsupseteq S'_0$. It means that $S \sqsubseteq S_0 \sqsubseteq S^* \sqsupseteq S'_0 \sqsupseteq S'$, i.e. $S \parallel S'$.

In appendix B we provide the detailed version of the proof.

Now, since we know that our algebra is sound and complete for its intended interpretation, we can use it to define the specification of the file synchronizer.

4 Definition of conflicting commands

We define conflicting updates using minimal sequences provided by the update detector algorithm. Actually, we define conflicting *pairs of commands* since only two modifications can interfere with each other. Later, every command will be marked as a conflicting command if it is a member of a conflicting pair.

Let us consider two commands, $C_A(\pi) \in S_A$ and $C_B(\gamma) \in S_B$, where S_A is the minimal sequence which leads the original filesystem O to replica A and, similarly, S_B leads O to replica B.

Now C_A and C_B are conflicting commands iff

$$(C_B \notin S_A) \wedge (C_A \notin S_B)$$

and one of the following holds:

- $\neg(C_A(\pi); C_B(\gamma) \parallel C_B(\gamma); C_A(\pi))$ (they do not commute), *or*
- $C_A(\pi); C_B(\gamma) \equiv \mathit{break}$, *or* $C_B(\gamma); C_A(\pi) \equiv \mathit{break}$
  (they break every filesystem).

We write $C_1 \leftrightarrow C_2$ if C_1 and C_2 are conflicting commands. Clearly this relation is symmetrical, i.e., $C_1 \leftrightarrow C_2 \implies C_2 \leftrightarrow C_1$.

In appendix D we try to find a relation between this definition and another one found in the paper of Balasubramaniam and Pierce.

5 Algorithm for reconciliation

In this section we provide an algorithm for reconciliation. The reconciler algorithm takes the sequences leading from the original filesystem to the replicas $(S_1, S_2, \ldots S_n)$, and creates sequences of commands for each replica $S_1^*, S_2^*, \ldots S_n^*$ which make the filesystems as close as possible.

When creating the algorithm, we keep in mind the following:
a command $C \in S_1 \cup S_2 \cup \ldots \cup S_n$ should be propagated to replica R_i iff:

- C is not already applied to R_i
- there are no conflicts on command C
- there are no conflicts on any command which must precede C

Why do we need the last criterion? Consider the following case: in the original replica we had $O(\pi) = X_{File}$. We have modified replica A with the following commands: $\mathit{edit}(\pi, Y_{Dir}); \mathit{create}(\pi/p, W_{File})$. Replica B has been modified by the sole command $\mathit{edit}(\pi, Z_{File})$. Now $\mathit{edit}(\pi, Y_{Dir})$ and $\mathit{edit}(\pi, Z_{File})$ are conflicting commands, so we cannot apply $\mathit{edit}(\pi, Y_{Dir})$ to replica B. But that way, we cannot propagate $\mathit{create}(\pi/p, W_{File})$ to it, either.

A command C_1 must precede command C_2 iff $\neg(C_1; C_2 \parallel C_2; C_1)$, it originally preceded C_2, and they appear in the same sequence S_i.

Now, in order to preserve the order of commands in sequences $S_1, S_2, \ldots S_n$, we define the reconciler algorithm as follows:
To determine S_i^*:

1. Detect conflicting commands between sequences
2. Detect command which must follow conflicting commands
3. Omit these commands from all sequences
4. Omit commands in S_i from all other sequences
5. Define S_i^* as $S_1; S_2; \ldots S_{i-1}; S_{i+1}; \ldots S_n$

Or, more formally:

```
FOR every sequence $S_i$
  FOR each command $C \in S_i$
    FOR every sequence $S_j$
      IF $C$ should be propagated to replica $R_j$ THEN
        append $C$ to $S_j^*$.
```

We can get the sequences of conflicting commands using a similar algorithm. The combined reconciler produces two sequences for every replica: the first one (S_i^*) can be applied to the replica immediately, while the second one containing the conflicting commands (S_i^C) needs the user's or (other algorithms') help to be resolved.

See Appendix E for the test results of an implementation of the algorithm.

6 Conclusion

We have built an algebra of commands on filesystems. We have also proved that the algebra is sound and complete, that is, commands which are algebraically equivalent are also equivalent on real filesystems, and vice versa.

We have defined conflicts with the help of this algebra, and defined algorithms for update–detection and reconciliation. We have also implemented these algorithms and tested them on numerous cases (see Appendix E). The results have always been in accordance with what we expected from the philosophy "propagate every command possible on every replica". Because of that, using algebraical methods in file synchronization has turned out to be an effective solution.

We have also gained some insight into the provability of such theorems and algebras. The methods we have used in the proofs are likely be able to be used in proofs for other algebras as well, thus they might form the basis for a more general theory of synchronization.

7 Acknowledgements

I would like to thank my mentor, Norman Ramsey, for allowing me to work on this fascinating project, and for always being ready to discuss and help me to overcome whatever difficulties I ran into during my work. I also appreciate the constructive criticism of my tutor, Jenny Sendova, who was so kind as to read my drafts several times and tirelessly untangle my sentences fraught with overcomplicated grammar. I thank LeeAnn Tzeng for her remarks on the paper which helped to make it more accessible and understandable.

I would also like to thank all sponsors and members of staff of the Research Science Institute for making the research possible.

References

[1] Ramsey, Norman: *An algebraic approach to file synchronization.* May, 2000, technical report.

[2] S. Balasubramaniam — Benjamin C. Pierce: *What is a File Synchronizer?*

A Proof for the soundness theorem

Let us repeat our statement: We prove that for every two sequences of commands

$$S \equiv S^* \implies \forall F : SF = S^*F,$$
$$S \sqsubseteq S^* \wedge SF \neq \bot \implies \forall F : SF = S^*F.$$

Both statement is true if our axioms and inference rules are sound since we can only gain equivalent (or extending) sequences from axioms or by using inference rules.

First, we prove the soundness of our inference rules. The proof for the first two inference rules are quite similar. For the first one, we know that the intended interpretation of $S_1 \equiv S_2$ is true and our statement is that for every S, S', the interpretation of $S; S_1; S' \equiv S; S_2; S'$ holds. We know that for every filesystem F, $(S; S_1; S')F = S'(S_1(SF))$ and similarly $((S; S_2; S')F = S'(S_2(SF))$. We know that for every G, $S_1G = S_2G$ holds, and therefore as a special case $S_1(SF) = S_2(SF)$. Thus $\forall F : (S; S_1; S')F = ((S; S_2; S')F$. This is the interpretation of our statement.

For the second law, we know that the interpretation of $S_1 \sqsubseteq S_2$ is true and we show that in that case $S; S_1; S' \sqsubseteq S; S_2; S'$ is also true for all filesystems. We know that for every F if $S_1F \neq \bot$ then $S_1F = S_2F$. If $(S; S_1; S')F = \bot$ then the relation is clearly true (since it only refers to the points where the filesystem is not broken). In the other case, if $(S; S_1; S')F = S'(S_1(SF)) \neq \bot$, we know that $SF \neq \bot$ since a filesystem cannot be made non–broken again by applying commands to it. Now since $S_1 \sqsubseteq S_2$ is true, therefore either $S_1(SF) = \bot$ and the relation is true similarly to the case above; or $S_1(SF) \neq \bot$ and therefore $S_1(SF) = S_2(SF) \neq \bot$ and $S'(S_1(SF)) = S'(S_2(SF)) \neq \bot$.

The soundness proof of the third law is quite simple since if two sequences act the same on *all* filesystems then they act the same on a subset of the filesystems, too.

It remains to show the soundness of each individual law, but all the laws were derived from the definitions of the commands. For precise proofs, Appendix F shows an example.

Our theorem is proved.

B Proof of the completeness theorem

Let us repeat our theorem and introduce minimal sequences again:

$$\begin{aligned}&\forall S, S' :\\ &\qquad ((\forall G : (SG \neq \bot \wedge S'G \neq \bot) \implies SG = S'G)\\ &\qquad \wedge\\ &\qquad (\exists F : SF \neq \bot \wedge S'F \neq \bot \wedge SF = S'F)\\ &\qquad \implies S \parallel S'),\end{aligned}$$

where G and F refer to filesystems.

In the proof, by F, we mean a filesystem which satisfies the second condition. By G, we refer to any filesystem which satisfies $SG \neq \bot \wedge S'G \neq \bot$.

Now consider the set of sequences $\wp_S = \{S^* | S \sqsubseteq S^*\}$. Because of our preconditions, the sequence *break* is not in $\wp_S$ (if it was, that is, $S \sqsubseteq \mathit{break}$, $SF = \bot$ would hold since by definition $SF = (\mathit{break})F$ if $SF \neq \bot$).

Let S_0 be (one of) the shortest sequence(s) in $\wp_S$ and, similarly, S'_0 (one of) the shortest sequence(s) in $\wp_{S'}$. The following holds for these sequences:

$$(SG \neq \bot \wedge S'G \neq \bot \implies) \quad SG = S'G = S_0G = S'_0G \neq \bot.$$

Proof. Since $SG \neq \bot \wedge S \sqsubseteq S_0 \implies SG = S_0G$ and $S'G \neq \bot \wedge S' \sqsubseteq S'_0 \implies S'G = S'_0G$; and since $SG \neq \bot \wedge S'G \neq \bot$ by assumption $SG = S'G \neq \bot$ holds, we have $S_0G = SG = S'G = S'_0G \neq \bot$.

As a special case, we also know that $SF = S'F = S_0F = S'_0F \neq \bot$.

Now let us investigate these "minimal" sequences.

B.1 Investigating the command *edit*

We know that the command $edit(\pi, X_{Dir})$ commutes or collapses (i.e., a commuting or a simplifying law can be applied to it) with every command on its left side (see Laws 1, 2, 7, 10, 13, 6A, 19, 24, 25, 30_E, 32, 4A_E). We also know that $edit(\pi, X_{File})$ does the same on its right side (see Laws 1, 2, 7, 8, 9, 3B, 5B, 16, 17, 18, 30_E, 31). From Laws 1, 2 and 30_E we know that *edit*s commute amongst each other.

Therefore in a minimal sequence all $edit(\pi, X_{Dir})$ commands can be moved to the beginning, and all $edit(\pi, X_{File})$ commands to the end of the sequence. Since they commute amongst themselves, we can order the two groups alphabetically. Therefore if there were two commands on the same path, they would be neighbors and would be simplified by the algebraic laws. Therefore we can be sure that there are at most one $edit(\pi, X_{File})$ or $edit(\pi, X_{Dir})$ command on each path.

That way we also separated these commands from the other ones. Now we focus on the remaining part: on the *create* and *remove* commands.

B.2 Investigating *create* and *remove*

We will prove some lemmas about minimal sequences.

Lemma 0 If a sequence is minimal, then it cannot have any pairs of commands that match the left-hand sides of Laws 16–41, 3B, 4B, 5B. Also, if we apply any of the commutative laws (1, 2, 4A_E, 5A, 6A, 7–15), the resulting sequence is still minimal.

Proof. For the first part, the laws mentioned all give an equivalent shorter sequence, but by hypothesis, there is no equivalent shorter sequence. For the second part, the commutative laws do not change the length of a sequence.

Lemma 1 It is impossible that a command $remove(\pi)$ precedes (not necessarily as a neighbor) a command $remove(\pi_{\varepsilon sub})$ in a minimal sequence. (A parent cannot be removed before any descendant, or more precisely, $\nexists i, j : i < j \wedge S[i] = remove(\pi) \wedge S[j] = remove(\pi') \wedge \pi \preceq \pi'$.)

$\cdots$		$S[i] : remove(\pi)$					$S[j] : remove(\pi_{\varepsilon sub})$	$\cdots$

Proof. By contradiction; we assume there is such an i and j, and we show that implies $S \sqsubseteq break$.

We show the contradiction by induction on $j - i$. The base case is $j = i + 1$. In this case, by Laws 28 and 29, $S[i]; S[i + 1] \sqsubseteq break$, and therefore $S \sqsubseteq break$.

For the induction step, we perform a case analysis on $S[j - 1]$.

- If it mentions a path that is disjoint with $\pi_{\varepsilon sub}$, we can swap it with $S[j]$ to get an equivalent sequence, and by the induction hypothesis and transitivity of equivalence, $S \sqsubseteq break$.
- Also by Lemma 0 (Laws 22, 23, and 33_E), it cannot be a non-disjoint create operation. (Note that there are no *edit* operations in this part of the sequence according to our preconditions.)
- If $S[j - 1] = remove(\hat{\pi})$, if $\hat{\pi} \preceq \pi_{\varepsilon sub}$, then by laws 28 and 29, $S[j - 1]; S[j] \equiv \{break\}$, and therefore $S \sqsubseteq \{break\}$. But if $\pi_{\varepsilon sub} \prec \hat{\pi}$, then by transitivity $\pi \preceq \hat{\pi}$, so the induction hypothesis applies, and again $S \sqsubseteq break$.

Lemma 2 A command $create(\pi)$ cannot precede a command $remove(\pi_{\varepsilon sub})$. *Proof.* This can be proved the same way, only the base case differs; now $create(\pi); remove(\pi_{\varepsilon sub})$ equals *break* or expands to *skip* by Law 22 and 33_E. The induction step is the same.

Lemma 3 A command $remove(\pi_{\varepsilon sub})$ cannot precede a command $create(\pi)$. *Proof.* This can be proved similarly, in this case the base step is $remove(\pi_{\varepsilon sub}); create(\pi)$ equals *break* or expands to *edit* (Law 27 and 34_E); in the induction step we can prove that if $S[j - 1] = create(\pi^{sup})$ precedes $S[j] = create(\pi)$, according to the induction hypothesis, $S \sqsubseteq break$ (we know that $\pi^{sup} \prec \pi$). Otherwise the commands would be able to be simplified (which contradicts the precondition that /(S/) is a minimal sequence) or commuted (and therefore the induction hypothesis still holds). (Laws 11, 12, 20, 21, 26, 27, 34_E.)

Lemma 4 A command $\mathit{create}(\pi_{\varepsilon sub})$ **cannot precede a command** $\mathit{create}(\pi)$**.** *Proof.* We obtain this result from Lemma 3: the base step is $\mathit{create}(\pi_{\varepsilon sub});\mathit{create}(\pi) \equiv (\sqsubseteq)\mathit{break}$ (Law 15); the induction step is the same.

Lemma 5 A command $\mathit{create}(\pi_{\varepsilon sub})$ **cannot precede a command** $\mathit{remove}(\pi)$**.** *Proof.* Our base step now is $S[j-1];S[j] = \mathit{create}(\pi_{\varepsilon sub});\mathit{remove}(\pi) \sqsubseteq \mathit{skip}$ or equals *break* according to Laws 33_E and 23. But in our induction step, we go in the opposite direction and perform analysis on $S[i+1]$. If $S[i+1] = \mathit{create}(\pi_{\varepsilon sub sub})$ follows $S[i] = \mathit{create}(\pi_{\varepsilon sub})$, the induction hypothesis applies, therefore $S \sqsubseteq \mathit{break}$. Otherwise the commands can be simplified or commuted (Laws 24, 20, 21, 22, 23, 33_E.)

Lemma 6 It is impossible that a command $\mathit{remove}(\pi)$ **precedes a command** $\mathit{create}(\pi_{\varepsilon sub})$**.** *Proof.* Now our base step is $\mathit{remove}(\pi);\mathit{create}(\pi_{\varepsilon sub}) \sqsubseteq \mathit{edit}$ or equals *break* (Laws 34_E and 26). In the induction step we examine command $S[i+1]$. If it is not commutable or simplifyable, it can be only $\mathit{remove}(\pi^{sup})$. In that case the induction hypothesis applies.

B.3 More lemmas on commands

Let us return to the *edit* commands to prove two additional lemmas.

Lemma E1 A command $\mathit{remove}(\pi)$ **cannot precede a command** $\mathit{edit}(\pi, X_{File})$**.** *Proof.* By contradiction; we assume there are such two commands in the sequence, $S[i] = \mathit{remove}(\pi)$ and $S[j] = \mathit{edit}(\pi, X_{File})$ where $i < j$. We use induction on $j - i$. The base step is when $j - i = 1$. Now, according to Law 24, $S \equiv \mathit{break}$ which contradicts our condition on S. For the induction step, we assume that $S \sqsubseteq \mathit{break}$ if $S[j-1] = \mathit{edit}(\pi, X_{File})$. Now we investigate $S[j-1]$. If it is not $\mathit{remove}(\pi_{sub})$, then a commuting (or simplifying) law can be applied to $S[j-1];S[j]$. That way $S[j-1]$ would be $\mathit{edit}(\pi, X_{File})$, and according to the induction hypothesis $S \sqsubseteq \mathit{break}$. If $S[j-1]$ is $\mathit{remove}(\pi_{sub})$, we have the same result by Lemma 1.

Lemma E2 A command $\mathit{create}(\pi)$ **cannot precede a command** $\mathit{edit}(\pi)$**.** *Proof.* The same; last step is made according to Lemma 2.

Lemma E3 A command $\mathit{remove}(\pi)$ **cannot follow a command** $\mathit{edit}(\pi, X_{Dir})$**.** *Proof.* This proof is also very similar to the above ones. The base step is $\mathit{edit}(\pi, X_{Dir});\mathit{remove}(\pi) \equiv \mathit{remove}(\pi)$ (Law 31). In the induction step we investigate command $S[i+1]$. If it is not $\mathit{create}(\pi_{sub})$, a commuting (or simplifying) law can be applied. If it is, we have $S \sqsubseteq \mathit{break}$ according to Lemma 5.

Lemma E4 A command $\mathit{create}(\pi)$ **cannot follow a command** $\mathit{edit}(\pi, X_{Dir})$**.** *Proof.* The same; last step according to Lemma 4.

Keeping in mind that we have all the *edit* commands at the ends of the sequence, it follows from Lemmas E1, E2, E3 and E4 that there cannot be an *edit* and another command on the same path.

Lemma E5 A command $\mathit{edit}(\pi, X_{Dir})$ **cannot precede a command** $\mathit{edit}(\pi, X_{File})$**.** *Proof.* Now we know that between these commands there are no commands on path π. We also know from Lemmas 1–6 that there is at most one *remove* or *create* command on each path. Therefore there cannot be a $\mathit{create}(\pi_{sub})$ command since we would have to remove it before modifying π to a file. Thus we could have moved $\mathit{edit}(\pi, X_{Dir})$ to the end of the sequence and simplified it with $\mathit{edit}(\pi, X_{File})$.

B.4 Conclusions

As a result of the lemmas we know that *there is at most one command on each path in a minimal sequence.*

Now we can prove that $C \in S_0 \iff C \in S'_0$ for any command C. (Without loss of generality, it is enough to prove that $C \in S_0 \implies C \in S'_0$.)

- $create(\pi, X) \in S_0 \implies create(\pi, X) \in S'_0$. *Proof.* By contradiction. First of all, we know that $F(\pi) = \bot$, as otherwise S_0 would break F, and that would contradict our assumption. Now assume that $create(\pi, X) \notin S'_0$. We have three cases. If there is no command on path π in S'_0, then $S'_0F(\pi) = \bot$, and therefore $S_0F(\pi) \neq S'_0F(\pi)$, that is, $S_0F \neq S'_0F$, which contradicts our assumption that $S_0F = S'_0F$. If there was a command *edit* or *remove* on path π, it would break F since $F(\pi) = \bot$ and that again contradicts the precondition.

- $remove(\pi) \in S_0 \implies remove(\pi) \in S'_0$. *Proof.* Now we can be sure that $F(\pi) \neq \bot$ and $S_0F(\pi) = \bot$. If (instead of *remove*) there was no command on π in S'_0, then $S'_0F(\pi) \neq \bot$ would hold. In case of *edit*, $S'_0F(\pi) \neq \bot$ would hold (contradicting $S_0F(\pi) = \bot$). *create* would break F since $F(\pi) \neq \bot$. Again, we have contradiction in all cases.

- $edit(\pi, X) \in S_0 \implies edit(\pi, X) \in S'_0$. *Proof.* Similarly we cannot have *create* or *remove* on path π in sequence S'_0 since $F(\pi) \neq \bot$ and $S'F(\pi) = SF(\pi) \neq \bot$. But now we might have no commands on π in S'_0: that way if $F(\pi) = X$, then $S_0F(\pi) = X$ and $S'_0F(\pi) = X$, too. But clearly $\neg(S_0 \parallel S'_0)$. To prove that this is also impossible, we need the first condition about all filesystems.

  Define H as $F\{\pi \mapsto Y\}$ where Y has the same type (file/directory) as $F(\pi)$. If $SF \neq \bot$, then $SH \neq \bot$ since no command breaks the filesystem because of the contents (not the type) of a file or directory. Also, $S'F \neq \bot \implies S'H \neq \bot$. Now, as a special case of $S_0G = S'_0G$ (note that $SH \neq \bot \wedge S'H \neq \bot$ holds) we know that $S_0H = S'_0H$. But $S_0H(\pi) = edit(\pi, X)H(\pi) = X$ and $S'_0H(\pi) = H(\pi) = Y$. We have a contradiction, therefore the lemma is proved.

This means that S_0 *and* S'_0 *contain the same commands;* they can be different only in the order of the commands.

Now we will show that they can be made exactly the same by the commuting algebraic laws.

We know that *edit*s can be moved to the ends of the sequences and ordered. Let us suppose they are already arranged this way. Since we know that the two sequences contain the same commands, these parts of the sequences are the same. Therefore we can omit them in the discussion. Now we focus on *create*s and *remove*s as we did above.

If there are two commands in a minimal sequence referring to comparable directories, according to the lemmas, they can only be

- $create(\pi) \ldots create(\pi_{sub})$
- $remove(\pi_{sub}) \ldots remove(\pi)$.

Therefore any two *create* or *remove* commands in a (minimal) sequence can be freely interchanged if they refer to incomparable directories, but they have a well–defined order otherwise. (Please keep in mind that we have no *edit* commands in these parts of the sequences.) That is, we have a partial order over the set of these commands. Note that we cannot have cycles in this order (since that way π would be equal to one of its descendants). Because of this, there is always a minimal element, which has no predecessors. Also, this ordering is the same on both sequences.

We prove that the commands can be ordered the same way by induction on the length of the sequences. If it is 1, the problem can be solved easily. If the length of the sequences are $i > 1$, we can choose a command from S_0 which has no preceding commands (i.e., a command which must precede it). It is sure that it has no preceding commands in S'_0. We can move this command to the front of the sequences. Now the rest of the sequences have length $i - 1$, therefore they can be ordered according to the induction hypothesis.

We obtain S_0^* by applying this method to S_0, and ${S_0'}^*$ by applying it to S_0'. Since they contain the same commands, and the resulting order is the same, S_0^* and ${S_0'}^*$ are *exactly the same.* Now we have $S \sqsubseteq S_0 \sqsubseteq S_0^* \equiv {S_0'}^* \sqsupseteq S_0' \sqsupseteq S'$, that is, $S \parallel S'$. Our theorem is proved.

C A solution for update-detection for a move–free algebra

First, we provide an example which explains why we chose to work on move–free algebra.

C.1 Moving and removing subtrees

When trying to determine the minimal sequence of commands which has been applied to the original filesystem O so that we get its present state A we should try to move or remove subtrees instead of (re)moving all files in it in order to gain the most simple sequence possible. We can achieve this by counting the files under a directory. If most of them has to be moved, we should we move the subtree first and then the rest of the files back to their original place (majority voting).

C.2 Problems with move

But, unfortunately, difficulties emerge when using such commands.

First of all, consider the following situation: we have files $X/1, X/2, X/3, X/4, X/5, X/6$ in filesystem O. In replica A, we have moved $X/1, \ldots X/4$ to $Y/1, \ldots Y/4$. In replica B, we have moved only $X/1, X/2$ under the new directory Y.

In replica A, without using commands on subtrees, we'd have *move* $X/1 \to Y/1, X/2 \to Y/2, X/3 \to Y/3, X/4 \to Y/4$. But using subtree–moving, we'll notice that using *move* $X \to Y, Y/5 \to X/5, Y/6 \to X/6$ would be shorter, so this is our sequence of commands. In replica B, *move* $X/1 \to Y/1, X/2 \to Y/2$ is the shortest way.

Now we want to choose commands from each sequence so that they would not cause a conflicting update if propagated on other replicas. It is easily provable that such subsequences of commands do not exist. We reached the conclusion that it is not always the shortest sequence that should be used to generate the interleaving; or, in other words, we should be able to change the sequences according to the algebraic laws before choosing. It would make the problem much more complicated since we have many possibilities here as well as when choosing the subsequences. Using *move*, we also have a very complicated relationship among the commands considering which two can commute and which two not. (Note that *move* is the only function which modifies *two* paths at the same time.)

Therefore, we decided to focus on a *move–free* algebra and delay detecting possibilities of simplification (subtree–detection) until after update–detection and reconciliation.

C.3 A simple algorithm for update–detection

With no *move*s and subtree–commands, a simple algorithm can be used for update–detection. It takes information from the filesystems and gives a minimal sequence of commands which will make one similar to the other.

Let P_O be all the paths in the original filesystem, and P_A all the paths in the current state of the replica. (We assume $O, A \neq \bot$.) For every $\pi \in P_O \cup P_A$, add the following command to sequence S:

- *create*(π, X) iff $O(\pi) = \bot$ and $A(\pi) = X \neq \bot$,
- *edit*(π, X) iff $O(\pi) \neq \bot$ and $A(\pi) = X \neq \bot$ and $X \neq O(\pi)$,
- *remove*(π) iff $O(\pi) \neq \bot$ and $A(\pi) = \bot$.

Then arrange the commands so that

- all $\mathit{edit}(\pi, X_{Dir})$ commands precede all other commands,
- all $\mathit{edit}(\pi, X_{File})$ commands follow all other commands,
- $\mathit{create}(\pi, X)$ commands precede $\mathit{create}(\pi_{sub}, Y)$ commands,
- $\mathit{remove}(\pi_{sub}, X)$ commands precede $\mathit{remove}(\pi, Y)$ commands.

This can be done by the method discussed below.

First, we move all $\mathit{edit}(\pi, X_{Dir})$ commands to the front, and then all $\mathit{edit}(\pi, X_{File})$ commands to the end of the sequence. Now for the middle part, we know the following:

Since there is only one command on each path, we have a partial ordering on the set of the commands (see section B.4) showing us which command must precede another one. Two commands can be interchanged if they refer to incomparable paths; otherwise, they are one of the following:

- $\mathit{create}(\pi) \ldots \mathit{create}(\pi_{sub})$
- $\mathit{remove}(\pi_{sub}) \ldots \mathit{remove}(\pi)$.

That is, every *create* must precede all other *create*s on its descendants and all *remove*s must precede *remove*s on their parents.

This ordering can be represented by a graph. This graph is a tree (or forest). It is worth noting that every tree of the forest is formed by the same commands, *remove*s or *create*s, since two different types of commands cannot show up at comparable paths.

Starting at the leaves of the trees, we move the leaves as close as possible to their parent. (They will form a row at the left side of the branch–command at the *remove*–trees and at the other side at *create*–trees.) Then we order them alphabetically and form a *group–command* from the leaves and the branch. This group–command will act like a single command since all of its members commute the same commands. Then we continue this method until every tree is grouped into one meta-group–command. We order these groups again. That way we gain an order of the commands. We will gain the same order for the same set of commands every time.

We know that $SO = A$ since because of the ordering it does not break the filesystem. Also, S is a minimal sequence. *Proof.* By contradiction. If S' is shorter than S, then we have a path φ on which we have a command in S but not in S' since all paths are different in S. Now we know that $SF(\varphi) \neq F(\varphi)$ but $S'F(\varphi) = F(\varphi)$. (Note that we have no superfluous commands in S since we gained it from the update detector algorithm.)

D Equivalence of definitions of conflicting updates

In this section we try to find a relation between a definition for conflicting updates found in the paper of Balasubramaniam and Pierce ([2]) and our definition.

D.1 Detecting conflicts using "dirtiness"

In the paper of Balasubramaniam and Pierce, an update–detection method is described which, when applied to filesystems O and A, gives a set dirty_A for which the following holds:

- $\pi \notin \mathit{dirty}_A \implies A(\pi) = O(\pi)$ holds where O is the original filesystem (Definition 3.1.1 in the paper of Balasubramaniam and Pierce).
- dirty_F is up–closed for any filesystem, that is, if $\pi \preceq \pi_{\varepsilon sub}$ and $\pi_{\varepsilon sub} \in \mathit{dirty}_F$, then $\pi \in \mathit{dirty}_F$ (Fact 3.1.3).

This set is a *safe estimate* of paths where updates have been made, that is, it may contain paths where the replica has not changed.

According to the paper, there is a conflict at path π iff $\pi \in \mathit{dirty}_A, \mathit{dirty}_B$ and $A(\pi) \neq B(\pi)$ and $(A(\pi) \neq X_{Dir}) \vee (B(\pi) \neq Y_{Dir})$. We write $\mathit{conflict}(\pi)$ if there are conflicting updates at path π according to this definition. (Definition 4.1.2)

D.2 Does a conflict at paths imply that there are conflicting commands?

Consider the following case. We have the original filesystem:

$$\begin{aligned} O = \{&\mathit{root} &&\mapsto X_{Dir};\\ &\mathit{root/dir} &&\mapsto Y_{Dir};\\ &\mathit{root/dir/file} &&\mapsto Z_{File}\} \end{aligned}$$

In replica A, we apply the following commands to O: $\mathit{remove}(\mathit{root/dir/file}); \mathit{remove}(\mathit{root/dir})$. In replica B, we use $\mathit{remove}(\mathit{root/dir/file})$. Now, according to the definitions in subsection D.1, $\mathit{root/dir/file} \in \mathit{dirty}_A$ and $\mathit{root/dir} \in \mathit{dirty}_A$. We know that $\mathit{root/dir/file} \in \mathit{dirty}_B$ and therefore $\mathit{root/dir} \in \mathit{dirty}_B$ (dirty_B is up–closed). Thus we have a conflicting command at path $\mathit{root/dir}$, because it is dirty in both replicas and $A(\mathit{root/dir}) = \bot$, so one of the filesystem entries at that path is not a directory.

If we investigate this case using sequences, there is no conflict. Since we know that $S_A = \mathit{remove}(\mathit{root/dir/file}); \mathit{remove}(\mathit{root/dir})$ and $S_B = \mathit{remove}(\mathit{root/dir/file})$, we can apply the second command in S_A to replica B because it does not conflict with any command in S_B. (We think this case is not a conflicting update, however, some file synchronizers, like *unison*, detect such an update here.)

Therefore, conflicting paths do not imply conflicting commands.

D.3 A conflict at commands implies a conflict at paths

We prove this theorem for two replicas.

We prove that if we have filesystems $O, A, B \neq \bot$ and we have the minimal sequences S_A and S_B and the dirty–sets dirty_A and dirty_B from the update–detectors, then if we have conflicting commands in the sequences we also have conflicts based on dirtiness.

The fact that we gained the sequences from update–detectors allows us to assume that there are no superfluous commands in the sequences, e.g. a command which leaves the filesystem in the same state.

Since we cannot distinguish between directories by their contents in the model of the paper of Balasubramaniam and Pierce, we have some preconditions.

Our theorem is true if:

- commands do not modify directories to directories (that is, $O(\pi) = X_{Dir} \implies \mathit{edit}(\pi, Y_{Dir}) \notin S_A$ or S_B) *and*
- all directories have the same contents (that is, for any F and G, if $F(\pi)$ is a directory and $G(\gamma)$ is a directory then $F(\pi) = G(\gamma)$);
- and there are no superfluous commands in S_A and S_B.

First of all, notice that $C_A(\pi) \leftrightarrow C_B(\gamma)$ can be true only if $\pi \preceq \gamma$ or $\gamma \preceq \pi$. (Otherwise they would commute.) Without loss of generality suppose the first one is true. Now we know: $C_A(\pi) \leftrightarrow C_B(\pi_{\varepsilon sub})$, where $\pi_{\varepsilon sub} = \gamma$.

Our theorem is that

$$\textbf{if } C_A(\pi) \leftrightarrow C_B(\pi_{\varepsilon sub}) \textbf{ then } \mathit{conflict}(\pi),$$

that is, there is a dirty–conflict at path π.

Proof. We know that $C_A(\pi) \in S_A$, therefore $O(\pi) \neq S_A O(\pi)$ where O is the original filesystem (remember that there are no superfluous commands). Hence $\pi \in \mathit{dirty}_A$ according to the definition of this set.

The same holds for $C_B(\pi_{\varepsilon sub})$, therefore $\pi_{\varepsilon sub} \in dirty_B$. This implies that $\pi \in dirty_B$ since $dirty_B$ is up–closed.

Now we need to show that $A(\pi)$ or $B(\pi)$ is not a directory. Suppose that both of them are directories. That is, $A(\pi) = S_A O(\pi) = X_{Dir}$. In other words ($S_A$ is minimal) $C_A(\pi)O(\pi) = X_{Dir}$ since there are no other command on this path. Thus $C_A(\pi)$ is $create(\pi, X_{Dir})$ or $edit(\pi, X_{Dir})$, but we can be sure that $O(\pi) \neq X_{Dir}$ because otherwise we would edit (modify) a directory to a directory, and by assumption we cannot do that.

We also know that $B(\pi) = S_B O(\pi) = X_{Dir}$ since we do not distinguish between directories. Since we know that $O(\pi) \neq X_{Dir}$, we must have a command on π in S_B, namely $C^*_B(\pi)$. This command can be either $create(\pi, X_{Dir})$ or $edit(\pi, X_{Dir})$, but it is equivalent to $C_A(\pi)$ since we must use *create* in both cases if $O(\pi) = \perp$ and *edit* if $O(\pi) = X_{File}$. But then $C_A(\pi) \in S_B$ since $C_A(\pi) = C^*_B(\pi) \in S_B$, which contradicts the definition of conflicting commands.

We also need to show that $A(\pi) \neq B(\pi)$. If so, similarly to the above, $C_A(\pi) = C^*_B(\pi)$ would hold.

Now we know that every condition of a dirty–conflict holds. Our theorem is proved.

E Implementation

We created an implementation based on the algorithms described in Section 5 and Appendix C.3. Its main purpose was to verify that the algorithms are implementable and they work as we expect them. The program is written in Perl and it runs under UNIX systems. It handles two replicas and does not modify the filesystems; it only detects updates and conflicts among commands. Then it provides the sequences of commands which should be applied to the replicas.

It uses no archives of the filesystems, therefore we also provide it the original version of the replicas. This is because the implementation was built to examine the algorithms, not to make a complete synchronizer. It also does not simplify the outcoming sequences with *move* commands or commands on subtrees.

When referring to the contents of directories, it looks at the writable flags of a directory. That is, the contents of two directories are different if one of them is writable for the program and the other one is not.

The brief description of its method is the following:

1. It creates flat representations of the original filesystem and the replicas $O(), A(), B()$.

2. Update–detection: it defines the minimal sequences S_A, S_B for which $S_A O = A$ and $S_B O = B$.

3. It orders these sequences using the method discussed in section C.3.

4. Reconciling: it detects conflicting commands in sequences.

5. It provides sequences S^*_A and S^*_B for reconciliation.

6. It provides a list of paths where conflicts occurred.

E.1 Equivalence of *edit* and *create* commands

To be able to determine whether two *edit* or *create* commands are equivalent, we introduced a *list of samples* in the program, which is a list of paths. When the program detects an $edit(\pi, X)$ or a $create(\pi, X)$ command as an update, it looks for a path γ in the list for which $A(\pi) = A(\gamma)$ (in case it is detecting updates at replica A). If it finds such a γ, it will attach its number in the list to the command. If not, it appends π to the list and attaches the new number.

That way, two *edit* or *create* commands are equivalent if and only if they are applied to the same path and they have the same sample–number.

E.2 Notation

The program uses the ordinary `directory/directory/file` notation for paths. For commands, it uses:

`EF0/path`	for	$edit(path, X_{File})$
`ED0/path`	for	$edit(path, X_{Dir})$
`CF0/path`	for	$create(path, X_{File})$
`CD0/path`	for	$create(path, X_{Dir})$
`RM/path`	for	$remove(path)$,

where `0` is the sample–number for which $A(Samplelist(\texttt{0})) = A(\texttt{path})$.

When marking conflicting commands, it uses '`<->A3`' if the command conflicts with the 3rd command in sequence S_A and '`==>A3`' if the command $S_A[3]$ must precede the current command but it already conflicts, so the current command cannot be applied.

E.3 Examples

In this section we provide some examples of how the program runs. In the first section of the output the program lists the flat representation of the filesystems (the original and the two replicas). The next section is the result of the detection of updates and conflicting commands. It lists the samples and the sequences of commands (S_A and S_B). Then sequences S_A^* and S_B^* follow which should be applied to replicas to propagate non–conflicting commands. In the last section it lists paths where there were conflicts.

E.3.1 Example 1

This example is the case discussed in section D.2.

```
File Synchronizer
  by Elod Csirmaz 21.07.2000
#==List of filesystems============
---Original Filesystem------------
FS-O/dir/
FS-O/dir/file
---Replica A----------------------
---Replica B----------------------
FS-B/dir/

#==Update-detection and conflicts=
---Edit Samples-------------------
---Sequence O->A------------------
0:RM/dir/file
1:RM/dir/
---Sequence O->B------------------
0:RM/dir/file

#==Reconciliation=================
---Sequence A->O'-----------------
---Sequence B->O'-----------------
1:RM/dir/

#==Paths of conflicting commands==
==================================
```

As we can see, a situation like this does not cause conflicting updates.

E.3.2 Example 2

This example shows a case when we have conflicts because a conflicting command must precede another command. `/dir/file` was modified in replica A.

```
File Synchronizer
  by Elod Csirmaz 21.07.2000
#==List of filesystems============
---Original Filesystem------------
FS-O/dir/
FS-O/dir/file
---Replica A----------------------
FS-A/dir/
FS-A/dir/file
---Replica B----------------------


#==Update-detection and conflicts=
---Edit Samples-------------------
0:FS-A/dir/file
---Sequence O->A------------------
0:EFO/dir/file <->B0
---Sequence O->B------------------
0:RM/dir/file <->A0
1:RM/dir/ ==>B0

#==Reconciliation=================
---Sequence A->O'-----------------
---Sequence B->O'-----------------
#==Paths of conflicting commands==
dir/
dir/file
==================================
```

Actually, in this case the synchronizer cannot do anything, because all commands conflicted.

F Soundness proofs for individual laws

In this section we provide examples of how algebraic laws can be proved. The method we use is to consider as many cases as necessary to be able to predict the result of each command in the law. For example, if the law contains paths π and π_{sub} then we will consider 22 cases; see below. (Note that we listed all cases considering how π_{sub} can relate to π and if they have children or a parent. Also, we considered the case when $F(\pi)$ has only one child and it is $F(\pi_{sub})$ since in that case, modifying $F(\pi_{sub})$ can make $F(\pi)$ childless.)

If the law contains an *edit* command, we also distinguish between files and directories. If we determine the result of the sequence of commands of the left side in each case and this is the same as that of the right side, then the law holds.

Now we provide an example for the proof of law 18. We will only investigate the left side of the law, since the right side always gives the broken filesystem.

Case 0
$F(parent(\pi)) = \bot$
$F(\pi) = \bot$
$childless(\pi)$
$F(parent(\pi_{sub})) = \bot$
$F(\pi_{sub}) = \bot$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 1
$F(parent(\pi)) = A_{File}$
$F(\pi) = \bot$
$childless(\pi)$
$F(parent(\pi_{sub})) = \bot$
$F(\pi_{sub}) = \bot$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 2
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = \bot$
$childless(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = \bot$

$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 3
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = \perp$
$childless(\pi)$
$F(parent(\pi_{sub})) = \perp$
$F(\pi_{sub}) = \perp$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 4
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{File}$
$childless(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = \perp$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 5
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{File}$
$childless(\pi)$
$F(parent(\pi_{sub})) = \perp$
$F(\pi_{sub}) = \perp$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 6
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$childless(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = \perp$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 7
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$childless(\pi)$
$F(parent(\pi_{sub})) = \perp$
$F(\pi_{sub}) = \perp$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 8
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = \perp$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 9
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = o_{File}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = x_{Dir}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 10
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = o_{Dir}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = x_{Dir}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 11
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = o_{Dir}$
$children(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = x_{Dir}$
$children(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 12
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$F(parent(\pi_{sub})) = \perp$
$F(\pi_{sub}) = \perp$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 13
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$F(parent(\pi_{sub})) = A_{File}, parent(\pi_{sub}) \neq \pi$
$F(\pi_{sub}) = \perp$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 14
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$F(parent(\pi_{sub})) = A_{Dir}, parent(\pi_{sub}) \neq \pi$
$F(\pi_{sub}) = \perp$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 15
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$F(parent(\pi_{sub})) = A_{Dir}, parent(\pi_{sub}) \neq \pi$
$F(\pi_{sub}) = o_{File}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$F(parent(\pi_{sub})) = A_{Dir}, parent(\pi_{sub}) \neq \pi$
$F(\pi_{sub}) = x_{Dir}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 16
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$F(parent(\pi_{sub})) = A_{Dir}, parent(\pi_{sub}) \neq \pi$
$F(\pi_{sub}) = o_{Dir}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$F(parent(\pi_{sub})) = A_{Dir}, parent(\pi_{sub}) \neq \pi$
$F(\pi_{sub}) = x_{Dir}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 17
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$F(parent(\pi_{sub})) = A_{Dir}, parent(\pi_{sub}) \neq \pi$
$F(\pi_{sub}) = o_{Dir}$
$children(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$
$F(parent(\pi_{sub})) = A_{Dir}, parent(\pi_{sub}) \neq \pi$
$F(\pi_{sub}) = x_{Dir}$
$children(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 18
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$, only one
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = o_{File}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$, only one
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = x_{Dir}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 19
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$, only one
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = o_{Dir}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$, only one
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = x_{Dir}$
$childless(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 20
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$, only one
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = o_{Dir}$
$children(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$:
$F(parent(\pi)) = A_{Dir}$
$F(\pi) = o_{Dir}$
$children(\pi)$, only one
$parent(\pi_{sub}) = \pi$
$F(\pi_{sub}) = x_{Dir}$
$children(\pi_{sub})$
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN

Case 21
BROKEN
After $edit(\pi_{sub}, x_{Dir})$:
BROKEN
After $edit(\pi_{sub}, x_{Dir})$ and $remove(\pi)$:
BROKEN