Theory Of File Synchronization

Előd P. Csirmaz
Mihaly Fazekas
Budakeszi, Hungary

under the direction of
Dr. Norman Ramsey
Harvard University

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 the 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 nonconflicting 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 to create 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, two phases of a synchronizer's task can be separated: the *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 about them.

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 $(S_1^*, S_2^*, \ldots, S_n^*)$ 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. But 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 errorfree 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 orders to it. 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 errorfree orders have the same effect* and apply them to the system.

The reason we may have two differently ordered sequences of commands that 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, a 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 notations on filesystems and paths.

A filesystem can be *broken* ($F = \bot$) 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), or 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.

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.

We write $childless_F(\pi)$ iff $F(\pi)$ has no descendants, i.e., $\forall \gamma : \pi \prec \gamma \implies F(\gamma) = \perp$.

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 thought about 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 movefree algebra.

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:

$$
\begin{array}{ll}
 & create(\pi, X)F = \\
F\{\pi \mapsto x\}, & \text{iff } (F(\pi) = \perp) \wedge (F(parent(\pi)) = Y_{Dir}); \\
\perp, & \text{otherwise.} \\
 & edit(\pi, X_{Dir})F = \\
F\{\pi \mapsto X_{Dir}\}, & \text{iff } (F(\pi) \neq \perp) \wedge (childless_F(\pi)); \\
\perp, & \text{otherwise.} \\
 & edit(\pi, X_{File})F = \\
F\{\pi \mapsto X_{File}\}, & \text{iff } (F(\pi) \neq \perp) \wedge (childless_F(\pi)); \\
\perp, & \text{otherwise.} \\
 & remove(\pi)F = \\
F\{\pi \mapsto \perp\}; & \text{iff } (F(\pi) \neq \perp) \wedge (childless_F(\pi)); \\
\perp, & \text{otherwise.} \\
 & break\ F = \\
\perp, & \text{for every } F.
\end{array}
$$

where $parent(\pi)$ is the path which immediately precedes π, that is, for some name $\mathbf{p}$, $parent(\pi)/\mathbf{p} = \pi$. 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 $S_1F \neq \perp$ 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.

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

Commuting laws

1. $edit(\pi, X); edit(\pi_{sub}, Y) \equiv edit(\pi_{sub}, Y); edit(\pi, X)$
2. $edit(\pi_{sub}, Y); edit(\pi, X) \equiv edit(\pi, X); edit(\pi_{sub}, Y)$
4A$_E$. $create(\pi_{sub}, Y); edit(\pi, X_{Dir}) \sqsubseteq edit(\pi, X_{Dir}); create(\pi_{sub}, Y)$
5A. $edit(\pi, X_{Dir}); remove(\pi_{sub}) \equiv remove(\pi_{sub}); edit(\pi, X_{Dir})$
6A. $remove(\pi_{sub}); edit(\pi, X_{Dir}) \equiv edit(\pi, X_{Dir}); remove(\pi_{sub})$
7. $edit(\pi, X); edit(\varphi, Y) \equiv edit(\varphi, Y); edit(\pi, X)$
8. $edit(\pi, X); create(\varphi, Y) \equiv create(\varphi, Y); edit(\pi, X)$
9. $edit(\pi, X); remove(\varphi) \equiv remove(\varphi); edit(\pi, X)$
10. $create(\varphi, Y); edit(\pi, X) \equiv edit(\pi, X); create(\varphi, Y)$
11. $create(\pi, X); create(\varphi, Y) \equiv create(\varphi, Y); create(\pi, X)$
12. $create(\pi, X); remove(\varphi) \equiv remove(\varphi); create(\pi, X)$
13. $remove(\varphi); edit(\pi, X) \equiv edit(\pi, X); remove(\varphi)$
14. $remove(\varphi); create(\pi, X) \equiv create(\pi, X); remove(\varphi)$
15. $remove(\pi); remove(\varphi) \equiv remove(\varphi); remove(\pi)$

Breaking laws

3B. $edit(\pi, X_{File}); create(\pi_{sub}, Y) \equiv break$
4B. $create(\pi_{sub}, Y); edit(\pi, X_{File}) \equiv break$
5B. $edit(\pi, X_{File}); remove(\pi_{sub}) \equiv break$
16. $edit(\pi, X); create(\pi, Y) \equiv break$
17. $edit(\pi_{sub}, X); create(\pi, Y) \equiv break$
18. $edit(\pi_{sub}, X); remove(\pi) \equiv break$
19. $create(\pi, X); edit(\pi_{sub}, Y) \equiv break$
20. $create(\pi, X); create(\pi, Y) \equiv break$
21. $create(\pi_{sub}, X); create(\pi, Y) \equiv break$
22. $create(\pi, X); remove(\pi_{sub}) \equiv break$
23. $create(\pi_{sub}, X); remove(\pi) \equiv break$
24. $remove(\pi); edit(\pi, X) \equiv break$
25. $remove(\pi); edit(\pi_{sub}, X) \equiv break$
26. $remove(\pi); create(\pi_{sub}, X) \equiv break$
27. $remove(\pi_{sub}); create(\pi, X) \equiv break$
28. $remove(\pi); remove(\pi) \equiv break$
29. $remove(\pi); remove(\pi_{sub}) \equiv break$

Simplifying laws

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

Laws for *break*

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

Remaining pairs (no substitution)

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

Table 1: Algebraic laws

A or B. Extension laws (that is, where the relation is $\sqsubseteq$, not $\equiv$) are marked with E. (See Table 1 for the laws.)

3.2 Inference rules

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

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

As we can see, our inference rules are mere substitutions of a part of the sequence to another sequence. If $S_1 \equiv S_2$ or $S_1 \sqsubseteq S_2$ is a law itself, we say that we applied that 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 an algebraic relation between sequences also means that the sequences act the same on filesystems. It can be proven that for every two sequences S and S^* of commands

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

The proofs for the two cases are very similar. We use induction on the number of times the inference rules were applied to the sequences. We obtain these results by showing the soundness of each individual law (which can be done by investigating numerous cases) and the inference rules.

The detailed proof can be found in 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 $S_1 \parallel S_2$ and say that 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,

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

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 were, that is, $S \sqsubseteq break$, $SF = \bot$ would hold since by definition $SF = (break)F$ 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 $S_0F = S'_0F$, so they modify F at the same points). And since we have to get the same result for every G, the sequences must contain the same commands on each path. 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 a more 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 made by the update detector algorithm. Actually, we define conflicting *pairs of commands* since only two modifications can interfere with each other. Later we will mark every command as a conflicting command if it was a member of a conflicting pair.

Consider two commands

$$C_A(\pi) \in S_A \quad \text{and} \quad C_B(\gamma) \in S_B,$$

where S_A is the minimal sequence that 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
- $C_A(\pi); C_B(\gamma) \equiv$ *break* or $C_B(\gamma); C_A(\pi) \equiv$ *break* (they break every filesystem).

We write $C_1 \leftrightarrow C_2$ if C_1 and C_2 are conflicting commands.

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 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. S_i^* contains all nonconflicting commands from $S_1, S_2, \ldots, S_n$ which have not already been applied to S_i.

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

- C is not already applied to R_i;
- there are no conflicts on command C; and
- 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 modified replica A with the following commands: $edit(\pi, Y_{Dir})$; $create(\pi/p, W_{File})$. Replica B was modified by the command $edit(\pi, Z_{File})$. Now $edit(\pi, Y_{Dir})$ and $edit(\pi, Z_{File})$ are conflicting commands, so we cannot apply $edit(\pi, Y_{Dir})$ to replica B. But that way, we cannot propagate $create(\pi/p, W_{File})$ 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 algorithm as follows:

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

See Appendix E for a test result of the implementation of the algorithm.

6 Conclusion

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

We defined conflicts with the help of this algebra; then we defined algorithms for update-detection and reconciliation. Finally, we implemented these algorithms and tested it on numerous cases (see Appendix E). The results were always 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 turned out to be an effective solution.

We also gained some information about the provability of such theorems and algebras. The methods we used in the proofs are likely be able to be used in proofs for other algebras as well, so that we would be able to create a more general theory of synchronization.

7 Acknowledgements

I would like to thank my mentor Norman Ramsey for offering me such a fascinating project and for giving me a hand when all the theorems and proofs seemed to be incorrect. I also appreciate the constructive criticism of my tutor Jenny Sendova, who was so kind as to read through the drafts several times and tirelessly untangle my absolutely confusing complex sentences. I thank LeeAnn Tzeng for her remarks on the paper which helped me a lot to make it more understandable.

Also I would like to thank all sponsors and staff members 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

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

For the first two statements, the proofs are quite similar. Both relations between sequences mean that one sequence can be transformed to the other by applying laws to it. We will prove our theorem by induction on the number of laws which were applied to S to gain S^*. The base step is for zero laws, that is: $SF = SF$ if $S \equiv S$ or $S \sqsubseteq S \wedge SF \neq \bot$. It is clearly true. Now, if we know that the theorem is true for every sequence S^* obtained from S by applying $i-1$ laws to it, we show that the same holds for sequences obtained by applying i laws. Call such a sequence S^{**}.

For the first part, if we know that $S^* \equiv S^{**}$ by applying one law, then by definition we know that the sequences can be written in the following way: $S^* = S'; P; S'' \equiv S'; P'; S'' = S^{**}$. Now we know $S^*F = (S'; P; S'')F = S''(P(S'F))$ and $S^{**}F = (S'; P'; S'')F = S''(P'(S'F))$. But since $P \equiv P'$ is a law, we know that $PG = P'G$ for every G. Therefore $P(S'F) = P'(S'F)$ and thus $S''(P(S'F)) = S''(P'(S'F))$, that is, $S^*F = S^{**}F$. By the induction hypothesis $SF = S^*F$ and therefore $SF = S^{**}F$.

The second part (when $S^* \sqsubseteq S^{**}$ and $P \sqsubseteq P'$ holds) can be proved similarly. For the induction step we assume that $SF = S^*F$ and $S^*F \neq \bot$. Now, proceeding with the separation as we did above, $S'F \neq \bot$, since $S^*F = S''(P(S'F)) \neq \bot$ and applying commands could not redefine filesystem $S'F$ if it was broken. Because of this, P and P' act the same way on $S'F$, that is, $P(S'F) = P'(S'F)$; and since $S''(P(S'F)) \neq \bot$ therefore $S''(P'(S'F)) \neq \bot$, so now we know $S^*F = S^{**}F$ and $S^{**}F \neq \bot$.

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:

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

where G and F refer to filesystems.

In the proof, by F, we mean a filesystem that satisfies the second condition. By G, we refer to any filesystem that 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 were, that is, $S \sqsubseteq break$, $SF = \bot$ would hold since by definition $SF = break\, F$ 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:

$$\begin{aligned}&(SG \neq \bot \wedge S'G \neq \bot \implies) \\ &\quad SG = S'G = S_0G = S'_0G \neq \bot.\end{aligned}$$

We now 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 that match the left-hand sides of Laws 16–41, 4B, 3B, 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 $remove(\pi_{esub})$ in a minimal sequence.

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 π_{esub}, 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 nondisjoint 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 $\hat{\pi} \preceq \pi_{esub}$, then by laws 28 and 29, $S[j-1]; S[j] \equiv break$, and therefore $S \sqsubseteq break$. But if $\pi_{esub} \prec \hat{\pi}$, then by transitivity $\pi \preceq \hat{\pi}$, so the induction hypothesis applies, and again $S \sqsubseteq break$.

The following lemmas can be proved with a similar method.

Lemma 2 A command $create(\pi)$ cannot precede a command $remove(\pi_{esub})$.

Lemma 3 A command $remove(\pi_{esub})$ cannot precede a command $create(\pi)$.

Lemma 4 A command $create(\pi_{esub})$ cannot precede a command $create(\pi)$.

Lemma 5 A command $create(\pi_{esub})$ cannot precede a command $remove(\pi)$.

Lemma 6 It is impossible that a command $remove(\pi)$ precedes a command $create(\pi_{esub})$.

B.3 More lemmas on commands

Now we go back to *edit* commands. We will prove two additional lemmas.

Lemma E1 A command $remove(\pi)$ cannot precede a command $edit(\pi, X_{File})$.

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

The following lemmas can be proved similarly.

Lemma E2 A command *create*(π) cannot precede a command *edit*(π).

Lemma E3 A command *remove*(π) cannot follow a command *edit*(π, X_{Dir}).

Lemma E4 A command *create*(π) cannot follow a command *edit*(π, X_{Dir}).

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

Lemma E5 A command *edit*(π, X_{Dir}) cannot precede a command *edit*(π, 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 *create*(π_{sub}) command since we would have to remove it before modifying π to a file. Thus we could have moved *edit*(π, X_{Dir}) to the end of the sequence and simplify it with *edit*(π, 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 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*(π, X) $\in S_0 \implies$ *create*(π, X) $\in S'_0$.

  Proof. By contradiction. First of all, we know that $F(\pi) = \perp$ otherwise S_0 would break F and that would contradict our assumption. Now assume that *create*(π, X) $\notin S'_0$. We have three cases. If there is no command on path π in S'_0 then $S'_0F(\pi) = \perp$ 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 $F(\pi) = \perp$ and that again contradicts the precondition.

- *remove*(π) $\in S_0 \implies$ *remove*(π) $\in S'_0$.

  Proof. It can be done similarly to the case of *create*.

- *edit*(π, X) $\in S_0 \implies$ *edit*(π, X) $\in S'_0$.

  Proof. Similarly we cannot have *create* or *remove* on path π in sequence S'_0 since $F(\pi) \neq \perp$ and $S'F(\pi) = SF(\pi) \neq \perp$. 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 $S_0 \not\parallel S'_0$. 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 $SF \neq \perp$ then $SH \neq \perp$ since no command breaks the filesystem because of the contents (not the type) of a file or directory. Also, $S'F \neq \perp \implies S'H \neq \perp$. Now, as a special case of $S_0G = S'_0G$ (note that $SH \neq \perp \wedge S'H \neq \perp$ 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 arranged this way. Since we know that the two sequences contain the same commands, these parts of the sequences is 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*(π) ... *create*(π_{sub}), or
- *remove*(π_{sub}) ... *remove*(π).

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. (Keep in mind that we have no *edit* commands in this part 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 that has no preceding commands (i.e., a command that must precede it). It is sure that it has no precedents in S'_0. Now we can move this command to the front of the sequences. Now the rest of the sequences have length $i - 1$; therefore, we can order them according to the induction hypothesis.

We obtain S_0^* by applying this method to S_0 and ${S'_0}^*$ from 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 movefree algebra

First, we provide an example which explains why we chose working on movefree algebra.

C.1 Moving and removing subtrees

When trying to determine the minimal sequence of commands which was 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 simplest sequence possible. We can achieve this by counting the files under a directory and how many of them should be (re)moved. If there are many files to be moved, we should move the subtree first and then the other files back to their 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, using the "simple way" for update detection, we would 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 the shortest sequence should not always 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 a lot of possibilities here as well as at choosing the subsequences. We can solve this problem by *delaying the subtree-detection* to the end of the update-detection and reconciliation.

We can also notice that only *move* made it necessary to use that complicated method in the "simple way" since this is the only function which modifies *two* paths at the same time. And as we mentioned above, that way we only gain *one* possible sequence. When choosing the subsequences, we would like to interchange the commands so as to choose the longest subsequence possible. But using *move*, we have a very complicated relationship among the commands considering which two can commute and which two not.

Therefore, we decided to focus on a *movefree* algebra and delay detecting possibilities of simplification (subtree-detection) until after the 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 that will make one similar to the other.

Let P_O be all the paths in the original filesystem, 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

- *create*(π, X) commands precede *create*(π_{sub}, Y) commands;
- all *edit*(π, X_{Dir}) commands precede all other commands;
- all *edit*(π, X_{File}) commands follow all other commands;

- $remove(\pi_{sub}, X)$ commands precede $remove(\pi, Y)$ commands.

This can be done by the method discussed below.

First, we move all $edit(\pi, X_{Dir})$ commands to the front and then all $edit(\pi, X_{File})$ 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 lemmas in Section B.2) 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:

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

That is, every *create* must precede all other *creates* on its descendants and all *removes* must precede *removes* on its parents.

This ordering can be represented by a graph. This graph is a tree (or forest). It is worth remarking that every tree of the forest is formed from the same commands, *removes* or *creates*, 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 with 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. It is worth remarking that we 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. For 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 $S'F(\varphi) = F(\varphi)$.

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, we can read about an update-detection method whose result when applied to filesystems O and A is a set $dirty_A$ for which the following holds:

- $\pi \notin dirty_A \implies A(\pi) = O(\pi)$ 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 $\pi \preceq \pi_{esub}$ and $\pi_{esub} \in dirty_F$ then $\pi \in dirty_F$ (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 $\pi \in dirty_A, dirty_B$ and $A(\pi) \neq B(\pi)$ and $(A(\pi) \neq X_{Dir}) \vee (B(\pi) \neq Y_{Dir})$. We write $conflict(\pi)$ 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:

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

In replica A, we apply the following commands to O: $remove(root/dir/file); remove(root/dir)$. In replica B, we use $remove(root/dir/file)$. Now, according to the definitions in subsection D.1, $root/dir/file \in dirty_A$ and $root/dir \in dirty_A$. We know that $root/dir/file \in dirty_B$ and therefore $root/dir \in dirty_B$ ($dirty_B$ is up-closed). Thus we have a conflicting command at path $root/dir$, because it is dirty in both replicas and $A(root/dir) = \perp$, 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 $S_A = remove(root/dir/file); remove(root/dir)$ and $S_B = remove(root/dir/file)$, 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, such as *unison*, detect such an update here.)

Therefore, unfortunately, conflicting paths do not imply conflicting commands.