



An Algebraic Approach to File Synchronization

The Harvard community has made this
article openly available. [Please share](#) how
this access benefits you. Your story matters

Citation	Ramsey, Norman and Elöd Csirmaz. 2001. An Algebraic Approach to File Synchronization. Harvard Computer Science Group Technical Report TR-05-01.
Citable link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:23853813
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

An Algebraic Approach to File Synchronization

Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University
Cambridge, USA

Előd Csirmaz

Mihaly Fazekas Secondary Grammar School
Budapest, Hungary

Abstract

We present a sound and complete proof system for reasoning about operations on filesystems. The proof system enables us to specify a file-synchronization algorithm that can be combined with several different conflict-resolution policies. By contrast, previous work builds the conflict-resolution policy into the specification, or worse, does not specify the behavior formally. We present several alternatives for conflict resolution, and we address the knotty question of timestamps.

1 Introduction

What is a file synchronizer? Suppose there are multiple replicas of a filesystem; perhaps you have one on a server, one on a computer at home, and one on a laptop. If you make different changes at different replicas, the replicas no longer contain the same information. A *file synchronizer* makes them consistent again, while preserving changes you made.

Not every set of replicas can be made consistent automatically. For example, if `src/hello.c` is created to say "Hello, world" on one replica and "Hello, Dolly" on another replica, it is not obvious how to choose one or the other. In cases like these, the file synchronizer needs a policy for *conflict resolution*. Reasonable people might differ about what constitutes a good policy; some alternatives appear in Section 6.

The behaviors of many synchronizers are not specified precisely; understanding how they detect and resolve conflicts can be difficult. Balasubramaniam and Pierce (1998) represents a major step forward; it specifies formal requirements for a file synchronizer, and it derives an algorithm from those requirements. This algorithm is implemented in the Unison file synchronizer.

Unison's specification is based on reasoning about states of the file system before and after synchronization. This state-based approach leads to an unnecessarily narrow view of conflicts. Balasubramaniam and Pierce (1998) actually builds the conflict-resolution

policy into the specification, making it unclear how to implement an interesting class of conflict-resolution policies.

We have taken a different approach to specification of file synchronizers: instead of reasoning about states, we reason about the operations that are performed at each replica. This paper makes the following contributions:

- We present an algebra of filesystem operations, together with algebraic laws that are helpful both for reasoning about file synchronization and for implementing synchronizers.
- We show that the laws are sound and complete with respect to a semantic model of file systems.
- We explain conflict detection and resolution in terms of our algebra, and we show that our technique detects essentially the same conflicts as the state-based technique of Balasubramaniam and Pierce (1998).¹
- We identify useful properties for conflict-resolution policies, including the *disconnected-repair* property, which a file synchronizer enjoys if a user can repair conflicts by making changes at a single replica. We also sketch how to express different policies using our algebra.

An algebraic approach to synchronization can simplify the specification, implementation, and user interface of a file synchronizer. It may also be possible to extend algebraic techniques to other synchronization problems, such as mail folders or PalmOS databases.

2 Formalizing the problem

We consider the synchronization of n replicas of a filesystem F , numbered F_1, \dots, F_n . Initially all replicas are identical: $F = F_1 = \dots = F_n$. At each

¹Our technique is actually slightly stronger. That is, if our technique detects a conflict, the state-based algorithm also detects a conflict, but there are cases in which the state-based algorithm detects a conflict that our model handles without conflict. These cases are uninteresting, however.

replica, users and programs perform operations on the filesystem. We write S_i for the sequence of operations performed at replica i . The task of the file synchronizer is to compute, for each replica, a sequence S_i^* that makes the replicas consistent and accounts for all the operations performed at each replica. If there are no conflicts, all replicas reach the same new state $F_{post} = S_1^*(S_1(F_1)) = \dots = S_n^*(S_n(F_n))$, where we take sequences of operations to act as functions on the state of a filesystem.

If order of operations didn't matter, we could simply compute $S = S_1 \cup S_2 \cup \dots \cup S_n$ and let $S_i^* = S \setminus S_i$. Because order does matter, however, we have to do more work. The problem comes from pairs of commands that don't commute; if $C_1; C_2$ has a different effect from $C_2; C_1$, not all orders are equivalent. The Introduction contains an example of such a pair of commands; if C_1 writes "Hello, world" and C_2 writes "Hello, Dolly", the last writer wins.

If operations were totally ordered, the problem might still be fairly simple; we would have to compute the list of all operations in the proper order, then arrange for the state of each replica to be as if that list of operations had been performed. Operations at an individual replica are totally ordered, but unfortunately we can't order operations between replicas. Even if we could guarantee consistency of timestamps, we wouldn't want to use timestamp ordering, because the agents (users and programs) that perform operations make decisions about what operations to perform by consulting only the states of their local replicas. Agents can't make decisions based on the results of operations performed at remote replicas, even if those actions have already taken place according to some global clock.

We frame the problem of file synchronization as first finding the set S of all operations that have been performed, then computing a useful subset of S such that within the subset, *all global orderings that are consistent with the local orderings have the same effect*. Using this subset, we can compute the sequences of commands S_i^* to be applied at each replica. In more detail, we can synchronize replicas in three steps:

1. *Update detection* examines each replica to determine the sequence of commands S_i that have been executed at the replica.
2. *Reconciliation* takes as many commands as possible from the sequences S_i and computes the sequences S_i^* to be executed at each replica.
3. *Conflict resolution* takes the leftover, "conflicting" commands and figures out what to do with them.

Our approach simplifies reasoning about all three steps, and in the third step it offers a significant advance over previous work: reasoning about commands makes it possible to devise several conflict-resolution strategies.

3 A precise model of filesystems

We model a hierarchical filesystem in which *paths* refer to files and directories. A path is simply a sequence of names. We use Greek letters for paths, most commonly π . Following Unix conventions, we use the / character to separate names in a path, and we write / for the empty path. 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 write $\pi \prec \gamma$ if π is a proper prefix of γ , that is, $\pi \preceq \gamma$ and $\pi \neq \gamma$. In filesystem terms, $\pi \prec \gamma$ means that π is an ancestor directory of γ . If $\pi \not\preceq \gamma$ and $\gamma \not\preceq \pi$, we say that π and γ are *incomparable*. It is a fundamental property of hierarchical file systems that operations taking place at incomparable paths are independent.

We write $parent(\pi)$ for the path that immediately precedes π . That is, if π is not empty, there is a name n such that $\pi = parent(\pi)/n$. The empty path has no parent.

We model a *working filesystem* F as a partial function mapping paths to files and directories. We write $F(\pi)$ to refer to the file or directory at path π in filesystem F . For the contents of a filesystem, we write

$F(\pi) = \text{FILE}(m, x)$	when path π contains a file with metadata m and contents x .
$F(\pi) = \text{DIR}(m)$	when path π contains a directory with metadata m .
$F(\pi) = \perp$	when filesystem F contains nothing at path π ; \perp is pronounced "missing."

Metadata may include permissions, ownership, modification time, etc., but the metadata of a directory explicitly does not include information about the directory's children; that information is encoded in F . We write $F(\pi) = X$ when we know $F(\pi) \neq \perp$ but we don't care if we're dealing with a file or a directory.

Our model also includes the *broken filesystem*, which we write $F = \perp$, pronounced "broken." A broken filesystem models the result of an erroneous command, e.g., deleting a directory with files under it. Broken filesystems don't occur in practice, because the operating system prevents users from breaking the filesystem.

We use a trivial lattice ordering of filesystems in which the broken filesystem is the bottom element. We write the lattice ordering $F_1 \sqsubseteq F_2$, pronounced " F_1 approximates F_2 ." This relation holds whenever $F_1 = \perp$ or when F_1 and F_2 are pointwise equal functions, i.e., $F_1 \neq \perp$ and $F_2 \neq \perp$ and $\forall \pi. F_1(\pi) = F_2(\pi)$.² The \sqsubseteq relation is a partial order, so two filesystems approximate each other if and only if they are equal.

²Readers familiar with denotational semantics should note that our ordering is not the ordering typically used for functions; in particular, if one working filesystem approximates another, they are identical.

To explain changes to working filesystems, we write $F\{\pi \mapsto X\}$ for the function that is like F , except it maps π to X .

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

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

4 An algebra of commands

What commands should we use to model operations on a filesystem? Because users must understand what a synchronizer is doing, our algebra of commands should be consistent with *users' mental models* of the actions they and their agents perform on the filesystem. Users might imagine performing operations like these:

$create(\pi, X)$	Create file or directory X at π .
$remove(\pi)$	Remove the file or directory that was at π .
$rename(\pi, n)$	Change the “base name” of a file or directory to n , while leaving it in the same place in the hierarchy.
$move(\pi, \pi')$	Move π to π' , also moving all descendants.
$derive(\pi)$	Change an existing file or directory, in a way that could be reproduced mechanically. Because the result can be reproduced, the operation need not say what the final state is. An obvious example is compiling a source to produce a binary.
$edit(\pi, X)$	Change an existing file or directory, leaving it in state X , in a way that can't be reproduced mechanically.

The distinction between *edit* and *derive* is useful because a user may wish to specify a behavior like “don't synchronize derived files.” We distinguish *create* from *edit* because although both operations have the same postcondition (file with new metadata and contents), they have different preconditions, so the distinction may help detect errors. Accordingly, we specify that to *create* an existing file, or to *edit* a nonexistent file, leaves the filesystem broken.

These high-level operations may be a good model for users, but they are not so good for deriving synchronization algorithms. We simplify.

- Conceptually at least, *move* can subsume *rename*, as it does in the Unix system (but not in early versions of DOS).
- *Derive* can't be distinguished from *edit* without knowledge about how files are derived. To avoid synchronizing derived files, we would be better off with

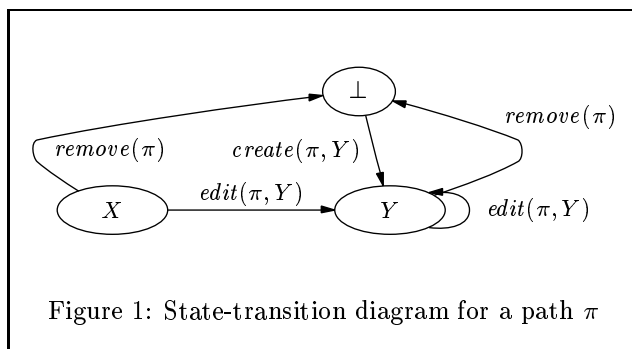


Figure 1: State-transition diagram for a path π

a more general mechanism for making files “invisible to the synchronizer.” We therefore drop *derive*.

- Finally, although it is not clear *a priori*, the *move* operation makes it more difficult to reason about synchronization. The crux of the problem is that the *move* operation affects two different locations in the filesystem, whereas the other operations affect only one. Accordingly, we replace $move(\pi, \pi')$ with the sequence $remove(\pi); create(\pi')$. The Unix synchronizer does the same. (A *move* can also be difficult to detect, but that is not sufficient reason to omit it from the algebra.)

Figure 1 shows how these operations change the contents of a filesystem at path π . Using the simpler operations simplifies synchronization but complicates a synchronizer's user interface. Section 6 explains how to recover a high-level view for interacting with users.

Precise definitions of the commands

We define the effect of each command as a function from filesystems to filesystems. Any command applied to a broken filesystem produces a broken filesystem. In the language of denotational semantics, every command is *strict* in the filesystem. Operationally, once a filesystem is broken, there is no way to fix it. Figure 2 gives the effects of commands on working filesystems. The command *break* is not one we expect to use during synchronization, but it helps us reason about errors. In particular, by showing that a sequence of commands is *not* equivalent to *break*, we can show those commands can be executed without error on at least one filesystem.

We are interested only in filesystems that satisfy the *tree property*: every parent must be a directory. Formally, if $\pi \prec \gamma$ and $F(\gamma) \neq \perp$ then $F(\pi) = \text{DIR}(m)$ for some m . The commands in Figure 2 maintain the tree property as an invariant.

The commands have another property that simplifies reasoning. Each command mentions at most one path π , and if a command is applied to a working filesystem, either it breaks the filesystem or it changes the filesystem only at π .

$$\begin{array}{lcl}
\text{create}(\pi, X) F & = & \begin{cases} F\{\pi \mapsto X\}, & \text{iff } F(\pi) = \perp \wedge F(\text{parent}(\pi)) = \text{DIR}(\dots) \\ \perp, & \text{otherwise} \end{cases} \\
\text{edit}(\pi, \text{DIR}(m)) F & = & \begin{cases} F\{\pi \mapsto \text{DIR}(m)\}, & \text{iff } F(\pi) \neq \perp \\ \perp, & \text{otherwise} \end{cases} \\
\text{edit}(\pi, \text{FILE}(m, x)) F & = & \begin{cases} F\{\pi \mapsto \text{FILE}(m, x)\}, & \text{iff } F(\pi) \neq \perp \wedge \text{childless}_F(\pi) \\ \perp, & \text{otherwise} \end{cases} \\
\text{remove}(\pi) F & = & \begin{cases} F\{\pi \mapsto \perp\}, & \text{iff } F(\pi) \neq \perp \wedge \text{childless}_F(\pi) \\ \perp, & \text{otherwise} \end{cases} \\
\text{break } F & = & \perp
\end{array}$$

Figure 2: Filesystem operations and their semantics

Algebraic laws

Our synchronization algorithm relies on proofs that different sequences of operations can have the same effects. We could construct such proofs by using the precise definitions of the commands in Figure 2, but it is awkward to reason directly about mathematical functions. This section presents the major technical contribution of this paper: a sound and complete proof system for reasoning about sequences of commands. This proof system appears in Table 1; it consists of algebraic laws that enable us to rewrite pairs of commands, plus inference rules for substitution and transitivity, which enable us to extend the rewriting to larger sequences.

We write commands in a sequence separated by semicolons. These sequences stand for functions from filesystems to filesystems, as described by this equation:

$$(C_1; C_2)(F) = C_2(C_1(F)).$$

We write S for a sequence of commands, and we write *skip* for the empty sequence of commands, i.e., the identity function on filesystems.

Although we want to reason about *equivalence*, the central relation of our algebra is not equivalence but *approximation*. To understand why, consider a sequence of two commands: one that creates a file, and a second that removes it. You might think this sequence is equivalent to *skip*:

$$\text{create}(\pi, X); \text{remove}(\pi) \stackrel{?}{=} \text{skip}.$$

Look again; the initial *create* operation is not safe on all file systems. If π is already present, or if π 's parent is not a directory, $\text{create}(\pi, S)$ breaks the filesystem. The correct relation between these two sequences is this:

$$\text{create}(\pi, X); \text{remove}(\pi) \sqsubseteq \text{skip}.$$

We pronounce $S_1 \sqsubseteq S_2$ as “ S_1 approximates S_2 ,” or sometimes “ S_2 is at least as good as S_1 .” The in-

tended interpretation is that we can use S_2 in place of S_1 without breaking more filesystems and without changing working outcomes. Frequently of course, two sequences are completely equivalent; we write $S_1 \equiv S_2$ as an abbreviation for $S_1 \sqsubseteq S_2 \wedge S_2 \sqsubseteq S_1$. Most of the laws in Table 1 do in fact use equivalence; laws using approximation are marked with the \sqsubseteq symbol.

We have organized Table 1 to show that we have considered all possible pairs of operations. There are 7 pairs involving *break*. These pairs lead to laws 37–43, which are consistent with Figure 2; once a filesystem is broken, no operation can fix it, and we know nothing about what happened before it broke.

There are 9 pairs of operations not involving *break*. Each such operation mentions exactly one path, and when we have a pair of paths π_1 and π_2 , there are four cases to be considered depending on the values of $\pi_1 \preceq \pi_2$ and $\pi_2 \preceq \pi_1$:

$\pi_1 \preceq \pi_2$	$\pi_2 \preceq \pi_1$	How we write π_1, π_2
T	T	π, π
T	F	$\pi, \pi/\pi'$
F	T	$\pi/\pi', \pi$
F	F	π, φ

These combinations account for 36 pairs of operations and paths, and for the laws numbered 1–36. Laws 3–6 are further split into D and F forms to account for the difference in semantics between directories and files. For example, law 5D says that making π a directory commutes with removing a descendant of π , but law 5F says that making π a file and then removing a descendant always causes an error.³ We summarize the proof system as follows:

³Either π originally had no descendants, in which case trying to remove one is an error, or it did have descendants, in which case turning it into a file (as opposed to a directory) is an error.

Commuting or approximating pairs

1. $edit(\pi, X); edit(\pi/\pi', Y) \equiv edit(\pi/\pi', Y); edit(\pi, X)$
2. $edit(\pi/\pi', Y); edit(\pi, X) \equiv edit(\pi, X); edit(\pi/\pi', Y)$
- 3D \sqsubseteq . $edit(\pi, DIR(m)); create(\pi/\pi', Y) \sqsubseteq$
 $create(\pi/\pi', Y); edit(\pi, DIR(m))$
- 4D \sqsubseteq . $create(\pi/\pi', Y); edit(\pi, DIR(m)) \sqsubseteq$
 $edit(\pi, DIR(m)); create(\pi/\pi', Y)$
- 5D. $edit(\pi, DIR(m)); remove(\pi/\pi') \equiv$
 $remove(\pi/\pi'); edit(\pi, DIR(m))$
- 6D. $remove(\pi/\pi'); edit(\pi, DIR(m)) \equiv$
 $edit(\pi, DIR(m)); remove(\pi/\pi')$
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. $remove(\pi); remove(\varphi) \equiv remove(\varphi); remove(\pi)$

Incorrect pairs

- 3F. $edit(\pi, FILE(m, x)); create(\pi/\pi', Y) \equiv break$
- 4F. $create(\pi/\pi', Y); edit(\pi, FILE(m, x)) \equiv break$
- 5F. $edit(\pi, FILE(m, x)); remove(\pi/\pi') \equiv break$
16. $edit(\pi, X); create(\pi, Y) \equiv break$
17. $edit(\pi/\pi', X); create(\pi, Y) \equiv break$
18. $edit(\pi/\pi', X); remove(\pi) \equiv break$
19. $create(\pi, X); edit(\pi/\pi', Y) \equiv break$
20. $create(\pi, X); create(\pi, Y) \equiv break$
21. $create(\pi/\pi', X); create(\pi, Y) \equiv break$
22. $create(\pi, X); remove(\pi/\pi') \equiv break$

$$\frac{S_1 \sqsubseteq S_2 \quad S_2 \sqsubseteq S_3}{S_1 \sqsubseteq S_3} \quad (\text{TRANSITIVITY})$$

23. $create(\pi/\pi', X); remove(\pi) \equiv break$
24. $remove(\pi); edit(\pi, X) \equiv break$
25. $remove(\pi); edit(\pi/\pi', X) \equiv break$
26. $remove(\pi); create(\pi/\pi', X) \equiv break$
27. $remove(\pi/\pi'); create(\pi, X) \equiv break$
28. $remove(\pi); remove(\pi) \equiv break$
29. $remove(\pi); remove(\pi/\pi') \equiv break$

Simplifying laws

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

Break is idempotent

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

Remaining pairs

- 6F. $remove(\pi/\pi'); edit(\pi, FILE(m, x))$
35. $create(\pi, X); create(\pi/\pi', Y)$
36. $remove(\pi/\pi'); remove(\pi)$

Non-pair laws

- BOTTOM. $break \sqsubseteq S$ for any S
REFLEXIVITY. $S \sqsubseteq S$ for any S

$$\frac{S_1 \sqsubseteq S_2}{S; S_1; S' \sqsubseteq S; S_2; S'} \quad (\text{SUBSTITUTION})$$

N.B. Paths π and φ are always incomparable. Where we write π/π' , π' is always nonempty.

Table 1: Proof system for the filesystem algebra

- Laws 1–2 and 3D–6D say what operations involving a directory and its descendant commute.
 - Laws 7–15 say that operations involving incomparable paths commute.
 - Laws 16–29 and 3F–5F say that operations which violate preconditions break the filesystem.
 - Laws 30–34 say when an operation can be combined with a previous operation.
 - Pairs 35, 36, and 6F, to which no laws apply, show significant constraints on non-breaking sequences: parents must be created before children; children must be removed before parents; and children must be removed before a directory can be made into a file.
 - Laws 37–43 say that any sequence containing *break* is equivalent to *break*.
 - The non-pair laws say that any sequence is at least as good as *break* and any sequence is at least as good as itself.
 - The inference rules say we can apply the laws within longer sequences, repeatedly if needed.
- Every pair law except law 3D can be used as a rewrite rule from left to right.

Soundness and completeness

The proof system in Table 1 is sound and complete. Informally, soundness says that any conclusion we draw using the proof system is safe, and completeness says

that any conclusion we draw using the underlying semantics can also (nearly) be drawn using the proof system.

Formally the soundness result is this:

$$S_1 \sqsubseteq S_2 \implies \forall F. S_1 F \sqsubseteq S_2 F.$$

The proof is straightforward, if a bit tedious, by induction on the proofs of judgments of the form $S_1 \sqsubseteq S_2$. We used automatic techniques to check the soundness of the algebraic laws.

Because of the possibility of commands that break the filesystem, our completeness result is not exactly what you might expect. We write $S_1 \parallel S_2$ (pronounced “ S_1 and S_2 have a common upper bound”) iff $\exists S : S_1 \sqsubseteq S \wedge S_2 \sqsubseteq S$. In other words, $S_1 \parallel S_2$ iff there is some sequence that is at least as good as both of them. In situations where neither S_1 nor S_2 breaks the file system, S_1 , S_2 , and the upper bound all have the same effect. Our completeness result shows that if the effect of S_1 approximates the effect of S_2 on every possible filesystem, the two sequences have a common upper bound:

$$(\forall F. S_1 F \sqsubseteq S_2 F) \implies S_1 \parallel S_2.$$

The implication is this: if there are two sequences of commands that have the same effect on every filesystem, we can find a third sequence that’s at least as good as either of the first two—and therefore has the same effect on whatever filesystems don’t break. We sketch the proof here; details will be relegated to an accompanying technical report.

We divide the proof into two cases. Suppose first that $\forall F. S_1 F = \perp$, that is, S_1 breaks all filesystems. By identifying the shortest prefix of S_1 that has this property, and by reasoning about the last operation in that prefix, we can show $S_1 \equiv \text{break}$, and $\text{break} \sqsubseteq S_2$ holds for any S_2 , so $S_1 \sqsubseteq S_2$ and S_2 is the common upper bound.

In the interesting case, $\exists F. S_1 F \neq \perp$, and $S_1 F \sqsubseteq S_2 F$ gives $S_1 F = S_2 F \neq \perp$. We define *minimal sequences* by considering the sets $\wp_S = \{S' \mid S \sqsubseteq S'\}$, and we let S^{min} be any sequence in \wp_S of minimal length. (The set is not empty because it contains S .) We show that $S_1^{\text{min}} F = S_2^{\text{min}} F \neq \perp$ and that *break* does not appear in either sequence. The proof of completeness has three main steps.

1. Because there is a filesystem that S_1^{min} and S_2^{min} do not break, no law mentioning *break* applies. Because they are of minimal length, no simplifying law applies. We conclude that in a minimal sequence, no path is mentioned more than once.
2. The sequences S_1^{min} and S_2^{min} must contain exactly the same set of commands. The key insight is that a command mentioning path π either breaks the filesystem or changes it only at π .

3. By applying commutative laws, we can rewrite S_1^{min} and S_2^{min} into a canonical sequence S . We use the following canonical ordering, which first orders commands by classes and then by pathname within class.

- (a) Commands of the form $\text{edit}(\pi, \text{DIR}(m))$, in any order determined by π .
- (b) Commands of the form $\text{create}(\pi, X)$, in preorder.
- (c) Commands of the form $\text{remove}(\pi)$, in postorder.
- (d) Commands of the form $\text{edit}(\pi, \text{FILE}(m, x))$, in any order determined by π .

To rewrite sequences into this form, we may apply law 4D, so the strongest result we can get is $S_1 \sqsubseteq S \sqsupseteq S_2$, not equivalence. The canonical sequence S may be *better* than S_1 and S_2 , that is, it may be correct on more filesystems, but whenever S_1 or S_2 works, S works and has exactly the same effect.

5 Using the algebra

We have applied our algebra to the three steps of file synchronization: update detection, reconciliation, and conflict resolution.

Update detection

Typical filesystems don’t keep logs of the operations that were performed on a filesystem; instead, we have to look at two states of a filesystem, F_i and F'_i , and find a minimal sequence of operations S_i such that $F'_i = S_i(F_i)$. We can do so by visiting all the non- \perp paths in each filesystem. As shown in Figure 1, by comparing $F_i(\pi)$ with $F'_i(\pi)$, we can decide whether a *create*, *remove*, or *edit* has taken place. We could conceivably infer an *edit* operation for each path that is populated in both filesystems; this strategy corresponds to the “trivial update detector” mentioned by Balasubramaniam and Pierce (1998). But this strategy makes the cost of synchronization proportional to the size of the filesystem, not the size of what has changed. To do better, we need to know which paths have identical values in both filesystems; no *edit* operations are needed for such paths.

Unfortunately, in typical use F_i represents the state of the filesystem at the last synchronization, F'_i represents the current state, and we may wish not to keep a copy of F_i available indefinitely.⁴ Even if we keep a copy, comparing contents of files may be expensive. Accordingly, file synchronizers typically

⁴Some operating systems, such as Plan 9, use write-once optical disks to make it cheap to reconstruct the state of a past filesystem (Thompson 1995), but such facilities are not common.

keep a *snapshot* of F_i , which is a copy of F_i that includes directory structure and metadata but omits the contents of files. That is, the snapshot saves $\text{FILE}(m, \perp)$ instead of $\text{FILE}(m, x)$. An alternative is to save $\text{FILE}(m, h(x))$, where h is a *fingerprinting* hash function (Broder 1993). The assumption is that in practice, we can avoid examining most contents because no operation changes the contents of a file without also changing its metadata. The details of exactly what metadata might change are subtle; for example, because Unix filesystems can rename files without changing their modification times, looking at modification time alone can miss updates. Looking at both modification time and inode number suffices; Section 3 of Balasubramaniam and Pierce (1998) has details.

Once we have decided on the *create*, *remove*, and *edit* operations that are needed, we can put these operations into canonical order. Our completeness theorem tells us that the canonical sequence is at least as good as what actually happened.

Reconciliation

Balasubramaniam and Pierce (1998) characterizes the requirements on a synchronizer using two slogans: (1) *propagate all non-conflicting operations* and (2) if operations conflict, do nothing. The value of our approach is that it enables choices about what to do at a conflict; our second slogan is therefore (2) *save conflicting operations for later resolution*.

We define conflicting operations using the minimal sequences found by the update detector. Consider two commands $C_i(\pi) \in S_i$ and $C_j(\gamma) \in S_j$, where $i \neq j$, and S_i and S_j are minimal sequences such that $F_i = S_i(F)$ and $F_j = S_j(F)$. We say $C_i(\pi)$ and $C_j(\gamma)$ are *conflicting commands* iff $(C_j \notin S_i) \wedge (C_i \notin S_j)$ and one of the following holds:

- $C_i(\pi); C_j(\gamma) \not\equiv C_j(\gamma); C_i(\pi)$, i.e., the commands do not commute.
- $C_i(\pi); C_j(\gamma) \equiv \text{break}$ or $C_j(\gamma); C_i(\pi) \equiv \text{break}$, i.e., the commands break every filesystem.

When C_1 and C_2 conflict, we write $C_1 \circ C_2$.

The *reconciler* takes the sequences S_1, \dots, S_n that are computed to have been performed at each replica. It computes sequences S_1^*, \dots, S_n^* that make the filesystems as close as possible. The idea of the algorithm is that a command $C \in S_i$ should be propagated to replica j (included in S_j^*) iff three criteria are met:

- $C \notin S_j$, i.e., C has not already been performed at replica j
- no commands at replicas other than i conflict with C
- no commands at replicas other than i conflict with commands that must precede C

A command C' must precede command C iff they appear in the same sequence S_i , C' precedes C in S_i , and they do not commute ($C'; C \not\equiv C; C'$).

To see why we have to worry about conflicts on commands that must precede C , consider this example. Suppose in the original filesystem $F(\pi) = \text{FILE}(m_x, x)$. We got two replicas by performing these commands:

$$\begin{aligned} F_1 &= (\text{edit}(\pi, \text{DIR}(m)); \text{create}(\pi/\text{n}, \text{FILE}(m_w, w)))F \\ F_2 &= \text{edit}(\pi, \text{FILE}(m_z, z))F. \end{aligned}$$

Commands $\text{edit}(\pi, \text{DIR}(m))$ and $\text{edit}(\pi, \text{FILE}(m_z, z))$ do not commute, so they are conflicting commands. Therefore we cannot apply command $\text{edit}(\pi, \text{DIR}(m))$ to replica 2. Because $\text{edit}(\pi, \text{DIR}(m))$ must precede $\text{create}(\pi/\text{n}, \text{FILE}(m_w, w))$, we cannot propagate the command $\text{create}(\pi/\text{n}, \text{FILE}(m_w, w))$ either.

Given our three criteria, the reconciliation algorithm must be equivalent to the following:

```

for  $i \in 1..n$  do
  make  $S_i^*$  empty
for  $i \in 1..n$  do
  for  $j \in 1..n$  do
    for every command  $C \in S_i$  do
      if  $C$  should be propagated to replica  $j$  then
        append  $C$  to  $S_j^*$ 

```

The algorithm is easily modified to compute the sets of conflicting commands S_i° as well as the sequences S_i^* .

6 Implementation

A prototype

To verify that our algorithms can be implemented and that they work as we expect, we have written a prototype implementation. The program is about 700 lines of Perl; when blank lines and comments are removed, under 400 lines remain. The program handles only two replicas, and it does not modify the filesystem; it simply computes the sequences S_1^* and S_2^* . Because it is a prototype, the program does not use a snapshot of the filesystem; instead we give it a complete copy of the original. The prototype also takes a simplified view of metadata; for example, the metadata for a directory is reduced to a single bit, which tells whether the program has permission to write the directory.

We have also started integrating our synchronization algorithm into the Unison synchronizer.

Scaling up to a real implementation

Although users may have a rich mental model of filesystem operations, it's easier to develop an effective proof system and a correct synchronization algorithm if we keep the algebra small. Still, there is no reason to make

the users suffer. *After* the synchronizer has computed the reconciling sequences S_i^* and the conflicting operations S_i° , we recommend introducing new operations to present these sequences to a user.

Collapsing ordered operations

In a minimal sequence, the only ordering constraints are those imposed by laws 3D, 21, and 29, as well as the pairs 6F, 35, and 36. Informally, parents must be created before children, and children must be removed before parents. We can eliminate ordering constraints by collapsing *create* and *remove* operations into operations on their parents. The collapsed operations might be called *create subtree*, *remove subtree*, and *edit into subtree*. The “collapsed form” of a minimal sequence is convenient because it enables us to forget about order, treating the sequence as a set. It should be helpful in a user interface. Not only is the subtree operation easier to understand, but if operations must be approved by users, as in the Unison synchronizer, the collapsed form makes it impossible for a user to approve an inconsistent set of operations (e.g., approving the creation of a file without also approving the creation of its parent directory).

Explicit *move*

We recommend introducing *move*, using the definition $move(\pi, \pi') = remove(\pi); create(\pi', X)$, where X is the contents of the original filesystem at π . Because the algebraic laws governing *move* are complex, we recommend that *move* be introduced only *after* reconciliation, to describe either actions to be taken or conflicting commands. Using *move* has three benefits.

1. *Performance*. If an agent at one replica has moved a file from π to π' , the instructions for performing the same action at other replicas need mention only the paths π and π' . If we treat the move operation as a deletion and creation, the instructions sent to other replicas must include the full contents of the file.

There are other solutions to this performance problem. In particular, if the synchronizer retains a “fingerprint” that uniquely identifies the contents of each file (Broder 1993), then one can build a transport layer that avoids sending the contents of any file whose contents are already available at another replica. But to realize the performance improvement, the synchronizer must be careful to send the *create* operation before the *remove* operation, lest contents that were available be discarded before they are needed. This ordering may conflict with orderings used in the user interface, e.g., lexicographic ordering by pathname, or ordering by type of operations at the convenience of the user.

2. *Retention of metadata*. We wish to be able to synchronize replicas that reside under different operating systems, such as Windows, Unix, and MacOS. Because each operating system has different metadata, it is in general impossible to preserve metadata when sending instructions between replicas under different operating systems. But there is an important special case, namely, a user running disconnected at F_1 wishes to restructure a directory whose contents contain metadata representable only at F_2 . If our algebra includes a *move* operation, we can propagate renaming operations from F_1 to F_2 without losing metadata that makes sense only at F_2 . If we do not have *move*, but must rely on *create*, we only send back to F_2 the results of a “best effort” to represent F_2 ’s metadata on F_1 , and we are likely to lose metadata like Windows access-control lists. (A formal characterization of “best effort” would be worthwhile, but it is beyond the scope of this paper.)
3. *Usability*. The most important reason to keep *move* is to reduce the cognitive burden on users. The Unison synchronizer, for example, first decides on a set of transactions, then asks its users to approve them.⁵ If a user is asked to approve a *move* operation, the user knows—from purely local information—that the contents of the renamed file will not be lost. But if the *move* is split into separate *create* and *remove* operations, these operations may be widely separated in the list of transactions; and a user wanting to be sure the *remove* is safe must inspect the entire list.

A *move* command also eliminates the possibility of an error in which a user approves the *remove* but not the corresponding *create*, resulting in loss of contents at one replica.

Resolving conflicts

By themselves, algebraic laws don’t tell us what to do with conflicting commands. Balasubramaniam and Pierce (1998) derives, from first principles, postconditions that completely determine a conflict-resolution policy. Our approach allows more freedom, which enables us to consider requirements for resolving conflicts. Three assumptions are common to both approaches:

- If there are no conflicts, the replicas are identical after synchronization.

⁵Unison’s transactions do not resemble the operations advocated in this paper. Instead, Unison offers three choices: make F_1 like F_2 , make F_2 like F_1 , or do nothing. Interestingly, Unison’s update-detection algorithm uses the operations in this paper (*remove*, *create*, *edit*, and *skip*), and it suggests a transaction based on what operation was performed at each replica. To help the user make a decision, Unison presents these operations in a simplified form. This form does not distinguish *create* from *edit*, and it collapses subtree operations as described above.

- Even in the presence of conflicts, the synchronizer preserves the knowledge of what changes were made by users.
- The knowledge of a human being is required to resolve conflicts. The human being is given the states F_i of the replicas, and the human being resolves conflicts by specifying either a new state F'_i for each replica or a sequence of commands S'^*_i to be performed at each replica.⁶ In a state-based approach, the synchronizer must be capable of establishing the desired state at each replica.

Balasubramaniam and Pierce (1998) resolves conflicts by ignoring conflicting commands. This policy preserves the information created at each replica, but that information is available *only* at the local replicas; *disconnected repairs* are impossible. We say that a synchronizer enjoys the *disconnected-repair property* if, no matter what the state of any replica, the following scenario is possible:

1. A synchronization is initiated (by human or other agency), and the synchronizer runs without human intervention.
2. The replicas are disconnected.
3. A human being resolves conflicts at one replica, leaving the other replicas unchanged. All information necessary to resolve conflicts must be present at each replica.
4. The replicas are reconnected, a second synchronization (“resynchronization”) is initiated, and it runs without human intervention.
5. The two replicas are identical.

If a person is to resolve conflicts, the synchronizer must tell the person what conflicts have occurred. An obvious mechanism would be to replicate information about conflicting commands, and to use a special user interface that would show this information and would enable people to resolve conflicts. But a special user interface may be unnecessary. Suppose instead that the synchronizer establishes a different postcondition: *After synchronization, all replicas are identical.* This means the synchronizer must somehow encode in the file system the information from both replicas. This encoding may be confusing, but the postcondition has compensating advantages.

- The user can determine the states of all replicas by examining a single replica.

⁶Human beings are of course free to write computer programs that compute new states or commands.

- The user need not remember what conflicts occurred at the most recent synchronization, because those conflicts manifest themselves as contents of the file system.
- The user can make disconnected repairs by mutating one replica until it reaches a desired state. Work can then proceed at that replica without resynchronization.

If results from synchronization are to be encoded in the filesystem, whenever there is a conflict at a pathname, the synchronizer must preserve, at all replicas, the distinct contents held by that pathname at each replica. To return to the example from the Introduction, `hello.c` might become `hello.c@server`, `hello.c@home`, and `hello.c@laptop`. But what happens to the original pathname, `hello.c`? We see three choices:

- *Remove the name.* This choice makes the conflict hard to overlook, but it has the disadvantage that if a pathname disappears, programs that rely on it may fail. In severe cases, a user may be unable to log in (e.g., a conflict occurs at `/bin/sh`).
- *Declare one replica the master*, and let the pathname reflect the value at the master replica. A permanent master could be designated, but it might also be useful to make the master the replica at which the synchronization was initiated. The advantage is that the master replica probably keeps working.
- *Let the replica with the latest modification time win.* This choice seems to offer the greatest risk—it may leave all replicas in unusable states—without any compensating advantages.

Metadata and modification times

Users have a right to expect that a synchronizer will propagate a file’s metadata as well as its contents. Most metadata can be propagated without difficulty, but because clocks at different replicas may show different times, propagating modification times can cause problems. Here are some requirements on timestamps:

1. *If the synchronizer thinks two replicas of a file are identical, those replicas should bear identical timestamps.* This requirement ensures that the files are treated as identical by other synchronization tools, by Make, by `find`, etc.
2. *When copying files from one replica to another, synchronization should not change the relative order of the timestamps.* This requirement preserves the correct behavior of Make. An early version of Unison used the time of synchronization as the modification time, sometimes leading Make to treat obsolete files as up to date.

3. *Timestamps at a single replica should be such that, if a user waits for one time unit to pass, then modifies or creates a file, that file will bear a modification time that is greater than the modification time of any other file at that replica.* This requirement is essential for Make to function correctly. If it is violated (e.g., because the system clock gets out of whack) the problem can be difficult to diagnose.
4. *The outcome of a synchronization should depend only on the state of the two file systems being synchronized, not on the time at which the synchronization takes place.* The argument for this requirement is that synchronization itself should not be seen as an operation on the filesystem, only as a way of propagating existing operations.

Requirements 2 and 3 are satisfied if this a more general condition holds: *If the user performs creation and modification operations at both replicas, and if these operations are totally ordered, then after the synchronizer runs, the timestamps on synchronized files respect this total order.* “Totally ordered” means not only ordered in real time, but ordered up to the ability of the local system to distinguish the actions. If a user changes two files 10 milliseconds apart, and time stamps have a granularity of one second, these two actions are not totally ordered.

The local clock provides an adequate total ordering for events at one replica, no matter what rate it runs at, provided it runs forward. The awful truth is that there is no way to tell when events at different replicas should be totally ordered, even when users take care to order them. As noted in Section 2, even if there is a global clock, we can’t rely on it, because we can’t know *post hoc* whether operations ordered in time were so ordered intentionally or accidentally.

If there is no consistent global clock, as is typically the case, the problems get worse; in the presence of clock skew, the conditions above cannot all be satisfied simultaneously. For example, if replica F_1 is running an hour ahead of replica F_2 , then changes to files modified within the last hour cannot be propagated to F_2 without either giving them different time stamps or violating the total ordering. We believe it is better to give them different time stamps.⁷ If the time skew is small, it may be even better to freeze synchronization for a few seconds, allowing the clock at F_2 to catch up with the latest modification time at F_1 . A formal study of synchronization in the presence of clock skew might yield more convincing recommendations.

⁷Even in this case, a synchronizer might well have to wait one tick at F_2 for every file synchronized, in order to respect the total order without creating any files “newer than now.”

7 Related work

Conflict detection

Balasubramaniam and Pierce (1998) phrases update detection in terms of *dirty sets*. Using our notation, the result of update detection applied to original filesystem F and replica F_i is a set $dirty_i$, which must satisfy two properties:

- $\pi \notin dirty_i \implies F_i(\pi) = F(\pi)$, i.e., clean files haven’t changed
- $\pi/\pi' \in dirty_i \implies \pi \in dirty_i$, i.e., if a path is dirty its parent is dirty

A dirty set is a *safe estimate* of paths where changes have been made; a good update detector computes the smallest possible dirty set. There is a *dirty-set conflict at path π* iff $\pi \in dirty_i \cap dirty_j$ and $F_i(\pi) \neq F_j(\pi)$ and either $F_i(\pi)$ or $F_j(\pi)$ is a file. (The specification in Balasubramaniam and Pierce (1998) ignores directory metadata, so all directories are considered identical. Unison’s implementation does not ignore directory metadata.)

We had expected our definition of conflicts, which uses *conflicting commands*, to be equivalent to the definition based on dirty sets. It turns out that our definition is strictly more liberal, but not in a terribly interesting way. The following example shows it is possible to have a dirty-set conflict without having conflicting commands. Let the original filesystem and the two replicas be given by these equations:

$$\begin{aligned}
 F &= \{ / \mapsto \text{DIR}(m), /d \mapsto \text{DIR}(m), /d/f \mapsto \text{FILE}(m_x, x) \} \\
 F_1 &= (\text{remove}(/d/f); \text{remove}(/d))F \\
 F_2 &= (\text{remove}(/d/f))F.
 \end{aligned}$$

The least dirty sets must be

$$\begin{aligned}
 dirty_1 &= \{ /, /d, /d/f \} \\
 dirty_2 &= \{ /, /d, /d/f \}
 \end{aligned}$$

N.B. $/d \in dirty_1$ because replica 1 changed at $/d$, but $/d \in dirty_2$ because $/d/f \in dirty_2$ and parents of dirty paths are dirty. We have a dirty-set conflict at $/d$ because it is dirty in both replicas and $F_1(/d)$ is not a directory.

Our algebra finds no conflict. $S_1 = \text{remove}(/d/f); \text{remove}(/d)$ and $S_2 = \text{remove}(/d/f)$, so there are no conflicting commands. In practice, we can safely apply $\text{remove}(/d)$ to replica 2, so we believe this example should be considered non-conflicting.

To show our algebra is more liberal, we show that whenever there are conflicting commands, there is a dirty-set conflict. For consistency with Balasubramaniam and Pierce, we assume that all directories have

the same metadata and write simply DIR for directories. We assume we have unbroken filesystems F, F_1, F_2 and we have the minimal sequences S_i and S_j and the dirty sets $dirty_i$ and $dirty_j$ from the update detectors. Finally, we assume that the minimal sequences do not contain unnecessary commands of the form $edit(\pi, \text{DIR})$. That is, because all directories have the same metadata, if $F(\pi) = \text{DIR}$ then the command $edit(\pi, \text{DIR})$ must not appear in S_1 or S_2 .

If two commands conflict, one path must precede the other, since otherwise the commands would commute. Without loss of generality, we number the replicas to choose $C_1(\pi) \in S_1$ and $C_2(\pi/\hat{\pi}) \in S_2$, where $\hat{\pi}$ may be empty, such that $C_1(\pi) \circ C_2(\pi/\hat{\pi})$. We prove there is a dirty-set conflict at path π .

Because each sequence S_i is of minimal length, we know that $F_1(\pi) \neq F(\pi)$ and $F_2(\pi/\hat{\pi}) \neq F(\pi/\hat{\pi})$. Therefore $\pi \in dirty_1$ and $\pi/\hat{\pi} \in dirty_2$. Because dirty sets are closed under the parent relation, $\pi/\hat{\pi} \in dirty_2$ means $\pi \in dirty_2$. What we have left to show is that $F_1(\pi) \neq F_2(\pi)$, and in particular either $F_1(\pi)$ or $F_2(\pi)$ is not a directory.

Suppose that $F_1(\pi) = F_2(\pi) = \text{DIR}$. Because S_1 is minimal, $C_1(\pi)$ is the only command in S_1 that mentions path π , and so $F_1(\pi) = (C_1(\pi)F)(\pi) = \text{DIR}$. We conclude that $C_i(\pi)$ must be either $create(\pi, \text{DIR})$ or $edit(\pi, \text{DIR})$. In either case we can be sure that $F(\pi) \neq \text{DIR}$ because otherwise $edit(\pi, \text{DIR})$ could be removed from S_1 , contradicting our assumptions. By assumption, $F_2(\pi) = \text{DIR}$, so there must be a command in S_2 that mentions π ; call it $C'_2(\pi)$. By similar reasoning $C'_2(\pi)$ must be either $create(\pi, \text{DIR})$ or $edit(\pi, \text{DIR})$, and since the replicas have the same initial and final states at π , in fact $C_1(\pi) = C'_2(\pi)$. But this forces $C_1(\pi) \in S_2$, which contradicts the assumption that $C_1(\pi) \circ C_2(\pi/\hat{\pi})$. Therefore $F_1(\pi)$ and $F_2(\pi)$ cannot both be directories.

Similar reasoning shows that $F_1(\pi) \neq F_2(\pi)$, and therefore we have a dirty-set conflict at π .

Other synchronizers

Space limitations preclude a thorough discussion of other synchronizers here. Commercial file synchronizers include Microsoft's *Briefcase* (Schwartz 1996; Microsoft 1998) and Leader Technologies' *PowerMerge*. Puma Technologies' *IntelliSync* solves a related problem: synchronizing various kinds of database files used in handheld and other computers (Puma a; Puma b). In addition to the Unison synchronizer (Balasubramaniam and Pierce 1998), there is an experimental synchronizer developed by the Rumor project (Reiher et al. 1996). Balasubramaniam and Pierce (1998) discusses some of these synchronizers, as well as connections to research in distributed file systems and databases. There is also the more recent Reconcile syn-

chronizer (Howard 1999).

8 Discussion

Balasubramaniam and Pierce (1998) specifies a file synchronizer by presenting preconditions and postconditions for the states of two filesystems before and after synchronization. Although these conditions completely determine a synchronization algorithm, we hope to have convinced you that other postconditions might be equally desirable, or possibly even more desirable. By reasoning about an *algebra of operations* instead of states, we have shown that there can be a *family* of specifications for file synchronizers, each of which could be considered correct. Different members of the family might offer different tradeoffs in their treatments of conflicting commands. Our algebraic approach illuminates the design space.

Our approach also shows additional promise, because *the filesystem algebra is not the only interesting algebra*. For example, the mail systems such as MH use filesystems to hold electronic mail. Directories represent mail folders, and files represent messages. File names represent message numbers. The message numbers themselves are not important. More precisely, although message numbers at an individual replica should not be changed gratuitously, it might be acceptable to have different message numbers at different replicas, and it might be acceptable if message numbers changed as a result of synchronization.

The mail-folder algebra corresponds not to filesystem operations but to mail-handling commands: `rmm`, which removes a message; `refile`, which moves a message between folders; and `inc`, which accepts delivery of new messages. Such commands assign message numbers and maintain internal invariants, e.g., the integrity of `.mh_sequences`. One may also see a rare *edit* operation, e.g., to patch botched headers, to reformat unreadable content created by Microsoft products, etc. A critical difference in the mail algebra is that messages should not be identified by pathname, but rather by contents. For messages that conform to RFC 822, the value of the Message-Id field can stand in for the contents. Our synchronization algorithm and proof techniques may nevertheless apply to this new algebra.

Existing synchronizers are either ill-specified (as are many of the commercial tools) or inflexible (Balasubramaniam and Pierce 1998). An algebraic approach seems to offer a natural and understandable path to specification and implementation of a file synchronizer, but the real potential advantages lie in two areas.

- Our algebraic approach leads not to a single synchronization algorithm, but to a family of possible algorithms. The approach seems especially useful for exploring different ways of resolving conflicts.

- We hypothesize that an algebraic approach may be useful for other synchronization problems, such as synchronizing mail folders, PalmOS databases, or other kinds of files with internal structure.

In the long run, it may even be possible to build a synchronizer that is parameterized by an algebra, an update detector, and a conflict resolver. One could extend such a synchronizer without having to prove the whole thing correct; instead, one could limit one's effort to proving the soundness of the algebraic laws and of the update detector.

Acknowledgments

Thanks to Benjamin Pierce for comments on this paper, and also for many stimulating discussions of file synchronization, especially during ICFP'99. Thanks to Tony Hoare for suggesting we focus on the refinement ordering. This work was supported by NSF grant CCR-0096069 and by the Research Science Institute, which is sponsored by the Center for Excellence in Education.

References

- Balasubramaniam, Sundar and Benjamin C. Pierce. 1998 (October). What is a file synchronizer? In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM-98)*, pages 98–108, New York. See the Unison home page at <http://www.cis.upenn.edu/~bcpierce/unison>.
- Broder, Andrei. 1993. Some applications of Rabin's fingerprinting method. In Capocelli, R., A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag.
- Howard, John H. 1999. Reconcile user's guide. Technical Report TR99-14, Mitsubishi Electronics Research Lab.
- Microsoft. 1998. Microsoft Windows 95: Vision for mobile computing. <http://www.microsoft.com/windows95/info/w95mobile.htm>.
- Puma. Designing effective synchronization solutions: A White Paper on Synchronization from Puma Technology. <http://www.pumatech.com/syncwp.html>.
- . A white paper on DSXtm Technology – Data Synchronization Extensions from Puma Technology. <http://www.pumatech.com/dsxwp.html>.
- Reiher, P., J. Popek, M. Gunter, J. Salomone, and D. Ratner. 1996 (June). Peer-to-peer reconciliation based replication for mobile computers. In *European Conference on Object Oriented Programming '96 Second Workshop on Mobility and Replication*.
- Schwartz, Stu. 1996 (May). The Briefcase—in brief. *Windows 95 Professional*. <http://www.cobb.com/w9p/9605/w9p9651.htm>.
- Thompson, Ken. 1995. The Plan 9 file server. In *Plan 9: The Documents*, pages 313–320. Murray Hill, New Jersey: Computing Sciences Research Center, AT&T Bell Laboratories.