# What's in Unison?

## A Formal Specification and Reference Implementation of a File Synchronizer

Benjamin C. Pierce
University of Pennsylvania

Jérôme Vouillon
CNRS

Technical Report MS-CIS-03-36
Department of Computer and Information Science
University of Pennsylvania

February 24, 2004

**Abstract**

A file synchronizer is a tool that reconciles disconnected modifications to a replicated directory structure. Trustworthy synchronizers are difficult to build, since they must deal correctly with both the semantic complexities of file systems and the unpredictable failure modes arising from distributed operation. On the other hand, synchronizers are often packaged as stand-alone, user-level utilities, whose intended behavior is relatively easy to isolate from the other functions of the system. This combination of subtlety and isolability makes file synchronizers attractive candidates for precise mathematical specification.

We present here a detailed specification of a particular file synchronizer called Unison, sketch an idealized reference implementation of our specification, and discuss the relation between our idealized implementation and the actual code base.

# 1 Introduction

The gradual shift from many users per computer to many computers per user has been accompanied by a significant increase in replication of important data. Replicating data ensures its availability during periods of disconnected operation, reduces latency during connected operation, and helps protect against loss due to system failures or user errors.

With this increase in replication comes, inevitably, the problem of how to propagate updates between replicas [4]. In tightly coupled systems, this issue can be addressed by conservative schemes that preserve "single-replica semantics." However, in systems whose parts must continue to operate when disconnected from each other, designers must often rely on *optimistic* strategies to achieve acceptable performance and availability. These schemes allow concurrent updates to replicated data that are later reconciled by automatically propagating non-conflicting changes to all replicas and detecting (and somehow resolving) conflicting changes.

The tools and system components that perform this sort of reconciliation, generically called *synchronization* (or *reconciliation*) technologies, come in many forms: distributed filesystems, operating systems, and databases, middleware layers, PDA hotsync managers, laptop synchronizers, etc. Our focus here is on the subcategory of *file synchronizers*: lightweight, user-level tools whose job is to maintain consistency between replicated directory structures. File synchronizers are similar in many ways to their big brothers, distributed filesystems, but the fact that they are user-level programs means that they run with no special access to operating system structures and that they are generally invoked explicitly by users, rather than working transparently in the background.

Engineering a file synchronizer is a challenging task. One reason for this is that a synchronizer must deal with all the semantic complexities and low-level quirks of real-world filesystems. Another is that file synchronization is an inherently distributed task, demanding robust operation in the face of a range of possible host and network failures. Finally, correctness is a critical issue, since misbehavior can corrupt arbitrary user data.

We have designed and implemented a file synchronizer called Unison.[1] From modest beginnings in the mid-90s, Unison has grown into a mature tool with a sizeable (tens of thousands) user community and high ambitions for portability, robustness, and smooth operation across different OS and filesystem architectures. An unusual feature of Unison's history is that the engineering of the system has proceeded in parallel with a serious effort to specify its behavior mathematically.

The main contribution of this paper is a simple, precise specification of Unison, phrased in terms that both implementors and users can understand. The core of this specification is based on an earlier specification by Balasubramaniam and Pierce [1], but extends and generalizes it in several respects, discussed in detail in Section 2. Our second contribution is a careful discussion of the "modeling gap" between our specification and reference implementation, on the one hand, and the actual Unison code base, on the other. Such gaps are a fact of life in specifications of full-blown systems programs, which are generally too complex and messy to formalize in full detail. They do not render the specification or reference implementation meaningless, but they do imply that the formal artifacts must be supplemented with significant engineering insight if they are to increase confidence in the actual code. We address these issues in Section 8. Our final contribution is a reference implementation, illustrating, in a few hundred lines of code, the main data and control structures found in our full-scale implementation. The complete reference implementation is presented in Appendix A. We have also carried out a formalized, machine-checked proof (using the Coq proof assistant) that the reference implementation satisfies the specification; we describe the outline of this proof.

We survey related work in Section 2. Sections 3 to 6 develop the core specification and its properties. Section 7 sketches the reference implementation and its proof of correctness, and Section 8 discusses the "modeling gap" between it and the real implementation. Section 9 shows how the basic model can be extended to handle file meta-data. Section 10 describes how the reference implementation is extended to deal with the possibility of failures during synchronization. Section 11 describes ongoing and future work. Appendix A presents the code for the reference implementation.

[1] `http://www.cis.upenn.edu/~bcpierce/unison`.

# 2 Related Work

At the implementation level, a large body of research on optimistic concurrency control in distributed databases, filesystems, and operating systems is related to our work [3, 4, 28, 14, 8, 5, 25, 15, 19, 21, 29, 13, 26, etc.]. The overall goals of all these systems (especially the distributed filesystems) are similar to those of Unison. The main differences are that the synchronization operations are intended to be transparent to the user (they are built into the OS rather than being packaged as user-level tools), and that most of them adopt a operation-based approach in which the synchronizer can see a trace of all filesystem activities, not just the final state at the moment of synchronization. Rumor [24] and Reconcile [9], on the other hand, are user-level synchronizers and share our static approach. Both of these systems go further than Unison in providing multi-replica synchronization, while Unison works with just two replicas; however, neither comes with a precise specification of its behavior. The overall design space of synchronization techniques (with emphasis on operation-based approaches) is discussed in an excellent survey by Saito and Shapiro [27].

At the level of specifications—our main focus here—previous work on synchronization is sparser. An early specification of a file synchronizer was reported by Balasubramaniam and Pierce [1]. This specification was later reformulated in an elegant algebraic style by Ramsey and Csirmaz [23]. The present paper extends and generalizes these earlier specifications in several respects. First, it presents a unified, "end to end" specification of the synchronizer's behavior from the user's point of view. In [1], the two major phases of synchronization—update detection and reconciliation—were specified separately, connected only by a notion of "*dirty* predicates"; this made the possible behaviors of the synchronizer difficult to calculate from the specification. Second, the present specification deals explicitly with the possibility of system failure during synchronization, through the notion of *maximality* (Section 5). The earlier specification simply gave the synchronizer a nondeterministic choice between a range of outcomes. Third, the present specification is based on a simpler and more abstract data structure (the *abstract filesystems* of Section 4). This enables us to present a simple specification of the synchronizer's core behavior (Section 5) and a more refined specification dealing with file metadata (Section 9) in a common notation framework. Finally, the reference implementation, the machine-checked proof of its correctness, and the discussion of the "modeling gap" between it and the actual Unison codebase are new.

Specification also plays a central role in the IceCube project at Microsoft Research [29, 13]. IceCube itself is a more ambitious effort than Unison, aiming not just to synchronize filesystems but to construct a "reconciliation middleware" platform that can be used by arbitrary (synchronization-aware) application programs. Its specification is parameterized by a collection of application *actions* with accompanying notions of *reorderability* and *conflict*. A major difference from our work, discussed further in the next section, is that IceCube is operation-based. Another operation-based reconciler, implemented by Molli and his co-workers [17, 10, 18], uses the idea of *operation transformation*, in which an operation that was only performed by the user on one of the replicas is propagated to the other by calculating an appropriately modified version taking into account the operations applied in the other replica.

The *session guarantees* [31] developed in the context of the Bayou replicated storage system [5, 32, 20, 6] are superficially similar to our "synchronization laws" in Section 5, but different in detail: rather than specifying the precise behavior of the synchronization operation, they take the whole replicated storage system as a black box and provide a layer on top of it that presents a more manageable overall semantics for application writers. A *session* in their sense is an abstraction for a sequence of read and write operations performed during the execution of an application. The idea of the guarantees ("read your writes," "monotonic reads," "writes follow reads," and "monotonic writes") is to present applications with a view of the database that is consistent with their own actions, even if they read and write from several, potentially inconsistent servers. The results of operations performed in a session should be consistent with the model of a single centralized server being read and updated concurrently by multiple clients.

A different area of specification work is the SyncML standard [30], being proposed by an industrial consortium of computer and PDA manufacturers (including Panasonic, Nokia, Eriksson, Starfish, Motorola, Palm, IBM, Lotus, and Psion). SyncML is a *protocol* specification rather than a *system* specification: it is mainly concerned with making sure that the devices engaging in a synchronization operation are speaking the same language, rather than with constraining the overall outcome of the operation. Another major difference is that SyncML deals only with flat record structures (mappings from atomic record-identifiers to records of simple data). Multiple data formats on different devices (related to our cross-platform goals) are supported, but only via application-specific mappings that are regarded as outside the purview of the specification.

# 3 Design Choices

Probably the most significant choice in Unison's design was making it a *user-level* tool, rather than building it into (or coupling it closely with) an operating system. Several factors motivated this choice. First, user-level synchronizers are easier to build and deploy than the synchronization components of distributed filesystems, since they stand above the rest of the filesystem and interact with it through the same APIs as other user programs; their relative isolation from other system components also eases the specification task. Second, user-level tools are better suited to the heterogeneous settings that we wanted to address: building an OS-level cross-platform synchronizer involves extending the kernels of at least two operating systems. Third, user-level synchronizers are an interesting subject for research: they are a very common category of tools in the real world, but one that has so far received little direct attention in the literature on distributed systems or formal specifications.

The choice to consider synchronizers as user-level programs imposes a very important constraint on their behavior. It is not generally possible (at least, in any portable fashion) for user programs to get access to a log or trace of updates to the filesystem. Hence, when the synchronizer runs, the only information it has available is the current contents of the replicas, plus whatever information it may have saved at the end of its last run. (Every user-level synchronizer maintains some record of the last synchronized state of the replicas, to avoid ambiguities between a creation in one replica and a deletion in the other.) This leads to a view of the synchronization task as being a matter of reconciling or merging the current *states* of the replicas. Operating-system-level synchronizers, by contrast, are often operation-based, leading to a view of synchronization as a question of merging the *traces* of actions on the different replicas. These different perspectives can lead to different conclusions about how synchronization should behave. For example, in the IceCube framework [29, 13], synchronization is defined as an operation on pairs of traces which must produce a single new trace incorporating as many actions as possible from the two input traces. That is, in IceCube, synchronization will *always* put the two replicas into the same state (i.e., the one resulting from applying this merged trace to their last common state), but some user updates may be undone in the process (when the two traces are partly incompatible). Unison, on the other hand, focuses on trying to reconcile states rather than traces; this leads to a different compromise, where we take our primary responsibility to be protecting user changes from being overwritten or "rewound" by the synchronizer, at the cost of sometimes leaving the replicas in different states after synchronization.

Having adopted a static view of the synchronization task, we can simplify the specification by taking a static view of filesystems as well. We represent filesystems as labeled trees, with internal nodes being directories, leaves being files, and labels being filenames. We then view a synchronizer as a simple function taking trees (representing the current states of the replicas) as inputs and producing trees (the synchronized replicas) as outputs. Of course, this view is an abstraction of reality, where synchronizers run with live filesystems that may be modified concurrently by other processes. To validate the abstraction, some work is required at the level of the implementation. This point is discussed further in Section 8.

Another issue concerns the treatment of the user interface. Like many file synchronizers, Unison provides a user interface that allows the user to examine *and override* the reconciler's recommendations before the replicas are changed. Our specification simply ignores the user interface—i.e., it assumes that the synchronizer is being run in "batch mode," where non-conflicting changes are automatically propagated and no action is taken for conflicting changes. In essence, we consider the user interface to be a completely separate tool that just happens to be compiled into the same binary as the core synchronizer and, for convenience, loosely integrated into its operation. Besides simplifying the specification, this view has a salutary effect on the code base, since it leads to an architecture where the complexities of the user interface code are kept rigorously separate from the complexities of the synchronization logic.

A final limitation is that our specification and reference implementation, like Unison itself, address only the case where exactly two replicas are being synchronized. The extension to multi-replica synchronization is discussed in Section 11.

## 4 Basic Definitions

To speak rigorously about synchronization, we need precise notations for filesystems and operations on them.

We give each filesystem node a *sort*, writing $\mathcal{S}$ for the set of all sorts. The core definitions and correctness conditions make no assumptions about $\mathcal{S}$; this abstract treatment helps keep things simple and facilitates a uniform treatment of file contents and metadata. For a basic synchronizer, we can take the sorts to be just "directory," written $\textsc{dir}$, and "file with contents $\mathtt{c}$," written $\textsc{file}(\mathtt{c})$. When we come to synchronizing filesystem metadata in Section 9, we will add sorts representing permission bits, modification times, etc., and refine atomic files into "pseudo-directories" with nodes of these new sorts as children.

Next, we assume given a set $\mathcal{N}$ of *abstract filenames* (or just *names*). Intuitively, these names can be thought of as ordinary filenames, but in Section 9 we will enrich $\mathcal{N}$ with special names ($\textsc{perms}$, $\textsc{modtime}$, etc.) for the children of the pseudo-directories representing files.

We use an abstract definition of filesystems that can be instantiated to both the simple and the more refined variants needed below simply by changing the sets $\mathcal{S}$ and $\mathcal{N}$. An *abstract filesystem* (or just *filesystem*) is an unordered tree whose nodes are labeled with sorts and whose edges are labeled with names. More formally, the set $\mathcal{F}$ of filesystems is the smallest set such that (1) $\perp \in \mathcal{F}$, and (2) $\textsc{node}(s, \mathit{chld}) \in \mathcal{F}$ for every $s \in \mathcal{S}$ and $\mathit{chld} \in \mathcal{N} \to \mathcal{F}$. (Note that we take the children of a directory to be a *total* function from names to filesystems and extend filesystems with an "absent" element $\perp$. We could, equivalently, omit $\perp$ and take $\mathit{chld}$ to be a partial function. The present formulation streamlines the development below by allowing creation, contents change, and deletion to be treated uniformly.) The variables $o$, $a$ and $b$ range over filesystems.

It is often convenient to talk about the contents of a filesystem at a *path*. Formally, a path is just a sequence of names. We write $\bullet$ for the empty path and $l/p$ for the path formed by prepending the name $l$ to $p$. We write $f@p$ for the *sub-filesystem* of $f$ rooted at $p$, defined as follows:

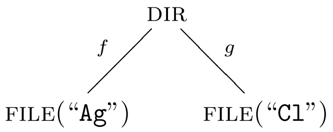
$$\begin{array}{lcl} f \mathbin{@} \bullet & = & f \\ \textsc{node}(s, \mathit{chld}) \mathbin{@} (l/p) & = & \mathit{chld}(l) \mathbin{@} p \\ f \mathbin{@} p & = & \perp \quad \text{otherwise.} \end{array}$$

We will need to make two kinds of comparisons between filesystems. First, we say that filesystems $f$ and $g$ *have the same root sort*, written $f \sim g$, if $f$ and $g$ are both $\perp$ or else if $f = \textsc{node}(s, \mathit{chld}_f)$ and $g = \textsc{node}(s, \mathit{chld}_g)$ for the same sort $s$. Second, we write $f = g$ to mean that filesystems $f$ and $g$ are structurally *equal* in the standard sense—i.e., $\perp = \perp$ and, if $\mathit{chld}_f(l) = \mathit{chld}_g(l)$ for all names $l$, we have $\textsc{node}(s, \mathit{chld}_f) = \textsc{node}(s, \mathit{chld}_g)$.

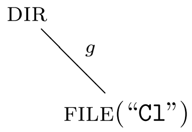
## 5 Core Specification

As we discussed in Section 3, our specification is phrased in completely static terms: filesystems are just trees, and the synchronizer is viewed as a function that maps pairs of trees to pairs of trees—i.e., we focus our attention on the states of the replicas before and after synchronization, ignoring the actual sequences of low-level filesystem operations that take them from one to the other (and, in the case of synchronization between remote hosts, the way the commands for performing these operations are sent over the network).

To make this picture precise, we need two refinements. First, we must deal with *create/delete ambiguities*. If the only inputs to the synchronizer are the current states of the two replicas, then, for example, there is no way for it to tell, given the following replicas, whether a new file $f$ (with contents "$\mathtt{Ag}$") has been created in replica $a$ or an existing file $f$ has been deleted from replica $b$.

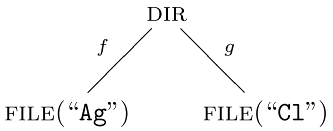


replica $a$



replica $b$

To know which change to propagate, the synchronizer needs to know the common original state from which the two replicas diverged. For example, if it knows the original state was



then it can see that $b$ is the one that changed. For this reason, all state-based synchronizers store some kind of *archive* between synchronizations, describing the last synchronized state of the replicas; this archive is provided as an additional input to the synchronizer.[2] The precise information stored in the archive varies from one synchronizer to another, but it is essentially just another filesystem.

The archive is not only useful for resolving create/delete ambiguities. It also gives us a clean definition of a "change" to a filesystem: we simply say that a path $p$ has *changed* in a replica $a$ (compared to an archive $o$) if $\neg(a@p \sim o@p)$—i.e., if the root of the subtree of $a$ at $p$ is not labeled with the same sort as the root of the subtree of $o$ at $p$ (or if one is $\perp$ and the other is not).

The second refinement follows from the fact that synchronizers may sometimes fail—and we want the specification to apply even in the case of failure. (We obviously cannot demand that synchronization should never fail, since we are talking about a distributed program. Just as obviously, we do not want to allow an arbitrary outcome if something fails in mid-synchronization.) Since failures may prevent the synchronizer from completing its work, a given set of inputs may in general lead to many possible outputs—i.e., we should regard the synchronizer as a *relation*, rather than a *function*, between its inputs and its outputs.

Formally, our specification is phrased as a set of properties that must be satisfied by every *run* of a correct synchronizer, where a run is a five-tuple $(o, a, b, a', b')$ of filesystems, representing an original synchronized state ($o$), the states of the two replicas just before synchronization ($a$, $b$), and the states of the replicas after synchronization ($a'$, $b'$). For brevity, we first define each property "locally"—i.e., we state what it means for the property to hold just between the root nodes of two given filesystems—and then demand that the property hold locally for all sub-filesystems; this avoids cluttering up the properties themselves with quantifications over all paths.

## Propagation of Changes

The first property a correct synchronizer must satisfy is that it must never overwrite a change made by the user. This requirement can be formalized as follows:

A run $(o, a, b, a', b')$ is said to *preserve user changes* (locally) if

$$\begin{array}{lcl} \neg(o \sim a) & \Longrightarrow & a' \sim a \\ \neg(o \sim b) & \Longrightarrow & b' \sim b. \end{array}$$

[2] Strictly speaking, it is both an input and an output, but for purposes of specification we can elide the latter, since the new archive can be calculated unambiguously from the new states of the two replicas. We discuss this point in more detail at the end of this section.

That is, if $o$ and $a$ have different sorts at the root (i.e., the user has changed the $a$ filesystem at the root since the last synchronization), then the synchronizer should leave the root sort unchanged in its result $a'$; and similarly for the other replica $b$.

For example, if the sort of $o$ is FILE(“`Ag`″) and that of $a$ is DIR, then the sort of the post-synchronization replica $a'$ must also be DIR. On the other hand, if $o$ and $a$ both have sort FILE(“`Ag`″), then $a'$ may have a different sort such as FILE(“`Cl`″).[3]

The next condition states that a correct synchronizer should change the replicas only by copying changes made by the user—it must never “make anything up.”

A run $(o, a, b, a', b')$ is said to *propagate only user changes* (locally) if

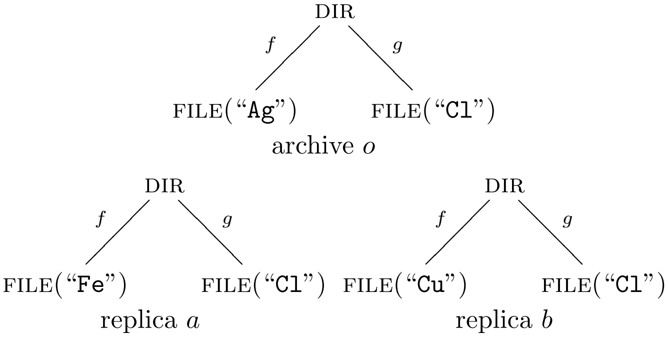
$$\begin{array}{lcl} \neg(a \sim a') & \Longrightarrow & b \sim a' \\ \neg(b \sim b') & \Longrightarrow & a \sim b'. \end{array}$$

That is, if the synchronizer changes the sort at the root of one of the filesystems (e.g., $a'$ differs from $a$), then the only value it may safely change it *to* is the sort from the root of the other filesystem ($a'$ should agree with the other replica, $b$).

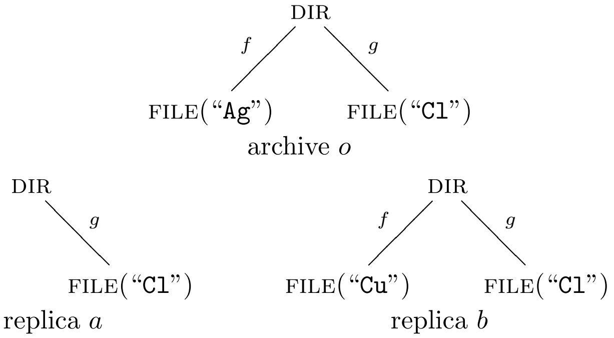
For example, if the sort of $a$ is FILE(“`Ag`″) and that of $a'$ is DIR, then the other original replica, $b$, must also have sort DIR.

## Treatment of Conflicts

The most delicate point in Unison's specification is the definition of conflicts. To begin with, we naturally expect that changing a file node in two different ways should result in a conflict,

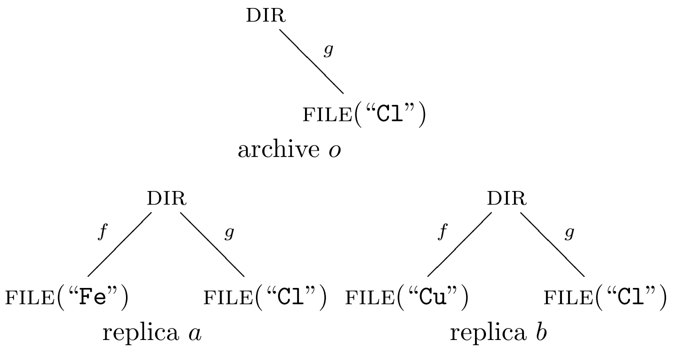


as should deleting a file in one replica and changing its contents in the other,



[3] This is a reassuring guarantee to be able to make to users; interestingly, though, there are synchronizers (e.g., the one built into the Microsoft Windows 2000 file sharing service) that do not obey this constraint—they simply overwrite the version with the earlier timestamp. This choice reflects an assumption that it is better to do *something* than *nothing* when faced with a conflict; this assumption is justified by the scale on which these systems are intended to be deployed (tens of thousands of replicas, rather than just two). If a file is changed simultaneously in Calcutta and San Francisco and the incompatible changes meet on a machine in Johannesburg at 2AM, to whom should the conflict be reported?

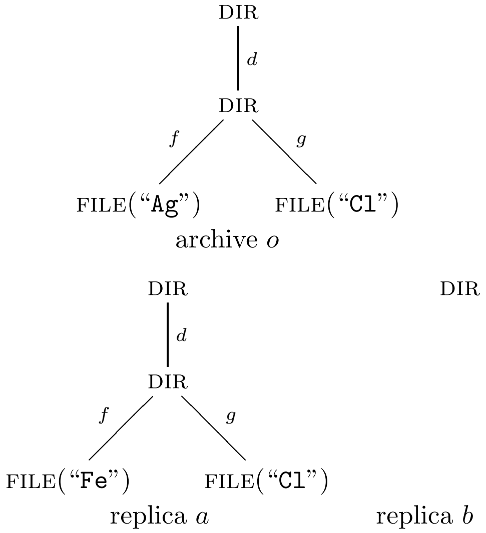
or creating the same new file with different contents.



archive $o$

replica $a$ replica $b$

The synchronizer's behavior in these cases is already completely determined by the first condition above: since both replicas have been changed by the user, neither can be changed by the synchronizer. However, because filesystems are tree-structured, there are some other cases to consider. For example, suppose that in one replica we change the contents of a file $d/f$, while in the other we delete the file's parent directory $d$.



archive $o$

replica $a$ replica $b$

Different synchronizers treat such situations in a variety of ways [1]. Unison adopts a conservative approach, defining this case as a conflict and refusing to take any action without explicit instructions from the user.

This leads us to the following definition of conflict.

> We say that $o$, $a$, and $b$ *conflict*, written *conflict*$(o, a, b)$, if
>
> $$\neg(a \sim b) \wedge \neg(o = a) \wedge \neg(o = b).$$

That is, two filesystems conflict if their roots are labeled with different sorts (e.g., one is DIR and one is $\perp$, as in the example above) and if both replicas has been changed *anywhere* (e.g., at the path $d/f$ in the example). Note the use of both kinds of "equality" in this definition: root-label equality between $a$ and $b$ and tree equality between $o$ and $a$ and between $o$ and $b$.

The third safety condition is now straightforward.

> A run $(o, a, b, a', b')$ is said to *stop at conflicts* (locally) if
>
> $$\mathit{conflict}(o, a, b) \quad \Longrightarrow \quad (a' = a) \wedge (b' = b).$$

That is, if a conflict is recognized at a path $p$, then the synchronizer is not allowed to change either replica anywhere below $p$. (In the example above, a conflict occurs at the path $d$, and the synchronizer cannot do anything at all. Note that, if we omitted the third condition, the other two conditions would prevent the synchronizer from doing anything at the paths $d$ or $d/f$, but would permit it to delete a sibling file $d/g$ from replica $a$. Some synchronizers actually exhibit this behavior, but we find it counterintuitive and feel that just signalling a conflict is safer.)

## Safety

We extend the local versions of the above properties to whole filesystems by saying that a synchronizer run $(o, a, b, a', b')$ has a given property *globally* if, for every path $p$, the run $(o@p, a@p, b@p, a'@p, b'@p)$ has the property locally. (Note that we need to consider paths that may not be valid in some of the filesystems involved. For example, if $o$ is $\perp$ and $a$ is not, we want to prohibit any modification of $a$ by the synchronizer. It is technically convenient simply to quantify over *all* paths.)

Now we can state the "do no harm" property that we want a synchronizer to obey.

**5.1 Definition:** A run $(o, a, b, a', b')$ is said to be *safe*, written $\mathit{safe}(o, a, b, a', b')$, if it satisfies the above properties (i.e., it preserves user changes, propagates only user changes, and stops at conflicts) globally.

## Maximality

The three safety conditions give a clear specification of what a synchronizer *may not* do, but they do not actually commit it to *doing* anything at all: the identity function is a safe synchronizer! Of course, we want synchronizer runs not only to do no harm, but also—when no failures occur—to do as much useful work as possible. This intention is captured by the notion of maximality.

We say that a run $(o, a, b, a', b')$ is *maximal* if it is safe and if it makes $a'$ and $b'$ as similar as possible—that is,

$$\begin{array}{l} \text{for all } a'' \text{ and } b'' \text{ with } \mathit{safe}(o, a, b, a'', b''), \\ \quad \text{for all } p, \\ \qquad a''@p \sim b''@p \implies a'@p \sim b'@p. \end{array}$$

If there is *any* safe run that leaves the sorts at path $p$ equal, then a maximal run must leave them equal. In other words, a maximal run is one that propagates all non-conflicting changes.

## Iterated Synchronization

Since we intend to run the synchronizer many times (interleaved with changes to the replicas), the archive is actually both an input *and an output* of each run: the "original state" for the next run of the synchronizer is calculated from the final state of the present run.

The calculation of the new archive would be trivial if it were not for the possibility of conflicts. If the synchronization is completely successful—i.e., if $a' = b'$—then we simply take this filesystem as the new archive. However, if $a' \neq b'$—i.e., if the present run encountered one or more conflicts—then we need to update the "last synchronized state" of just those files for which synchronization actually succeeded, while retaining the previous "last synchronized state" for paths that were left unequal.

More precisely, for each path where the new replicas $a'$ and $b'$ agree, we take their common value as the contents of this path in the archive; where they do not agree, copy the contents from the input archive $o$. Formally, this is expressed by taking a *run* to be a six-tuple of filesystems $(o, a, b, a', b', o')$. We extend the conditions above to the new type of runs in a trivial way (ignoring $o'$), and then add one final condition specifying how $o'$ is calculated from $a'$, $b'$, and $o$.

A run $(o, a, b, a', b', o')$ *returns a correct archive* (locally) if

$$\begin{array}{rcl} a' \sim b' & \Longrightarrow & o' \sim a' \\ \neg(a' \sim b') & \Longrightarrow & o' \sim o. \end{array}$$

For brevity in the rest of the paper, we will ignore this refinement and revert to the old five-place notion of runs.

# 6 Properties of the Specification

Before going any further, it is worthwhile to state a few properties of the specification, both as sanity checks on the definition and to deepen our understanding of its implications.

**6.1 Theorem [Laziness is safe]:** For every $o$, $a$, and $b$, the run $(o, a, b, a, b)$ is safe.

**Proof:** Observe that, for all $p$, the run $(o@p, a@p, o@p, a@p, a@p)$ satisfies all three local properties (this is easily verified in each case). □

**6.2 Theorem [Mirroring is a special case]:** For every $o$ and $a$, the run $(o, a, o, a, a)$ is maximal.

**Proof:** Clearly, the run $(o, a, o, a, a)$ is maximal if it is safe. To check safety, proceed as in the previous proof. □

**6.3 Theorem [Maximal runs are unique]:** For every $o$, $a$, and $b$, there are unique $a'$ and $b'$ such that $(o, a, b, a', b')$ is maximal.

**Proof sketch:** Suppose that we have two maximal runs, $(o, a, b, a', b')$ and $(o, a, b, a'', b'')$. Using the definition of maximal runs (in particular, the three local safety properties), we argue, for each $p$, that $a'@p \sim a''@p$; the symmetry of the definition gives us $b'@p \sim b''@p$. □

**6.4 Theorem [Success in the absence of conflicts]:** If, for every path $p$, we have $\neg \mathit{conflict}(o@p, a@p, b@p)$, then there exists a filesystem $c$ such that the run $(o, a, b, c, c)$ is maximal.

**Proof:** By induction on the size of $a$ and $b$. If both $a$ and $b$ are $\bot$, then we can take $c = \bot$, as $(o, \bot, \bot, \bot, \bot)$ is maximal. Otherwise, we know that $\neg \mathit{conflict}(o, a, b)$, that is, either $o = a$, $o = b$, or $a \sim b$. In the first two cases, we are done, as the run $(o, o, b, b, b)$ and $(o, a, o, a, a)$ are maximal. Otherwise, we apply the induction hypothesis on all $o@p$, $a@p$ and $b@p$ where $p$ is of the form $l/\bullet$. Thus, there exists a function $\mathit{chld} \in \mathcal{N} \to \mathcal{F}$ such that $(o@p, a@p, b@p, \mathit{chld}(l), \mathit{chld}(l))$ is maximal. Then we take $c = \textsc{node}(s, \mathit{chld})$ where $s$ is the common sort of $a$ and $b$. It is easy to check that $(o, a, b, c, c)$ is maximal. □

# 7 Reference Implementation

The specification described above provides a concise description of the intended behavior of a file synchronizer. But how do we know that a particular synchronizer implementation, e.g., Unison, actually satisfies the specification? For most of Unison's lifetime, our best answer to this query was, "We don't *know*; but the implementors all understood the specification and tried hard to write code that would satisfy it."[4] However, while writing this paper, we decided to try something a little more ambitious.

We extracted from the current Unison code base a *reference implementation*, keeping the essential control and data structures while abstracting away from the low-level complexities of real filesystems and networks; as in the specification, filesystems are represented as simple tree structures, while the network is ignored

[4] This is already a *major* improvement over "We didn't have any precise specification in mind, but we thought hard while coding and made decisions that seemed reasonable at the time"!

completely—we assume that both filesystems and the archive are given as inputs to a top-level synchronization function. We then proved that this implementation satisfies the specification.

The reference implementation is a straightforward—but, at about 450 lines of OCaml [16] code for the complete version, nontrivial—functional program. Like Unison itself, it works in three phases: update detection, reconciliation, and propagation of updates. For the sake of brevity, we present here a simplified core of the actual reference implementation, omitting file and directory metadata, symbolic links, and the modeling of failures and concurrent filesystem updates during synchronization. The extensions needed to handle metadata and failures are outlined in Sections 9 and 10.

**Update detection:** The function `buildUpdates` compares an archive `o` with a filesystem `a`. It returns a data structure `uia` describing the differences between the archive and the filesystem. This phase is performed separately for the two replicas. This separation is important because, in practice, the two replicas are often separated by a slow network, and we need to avoid transmitting the whole contents of either replica over the network. We achieve this by storing a copy of the archive on each host, doing update detection locally, and sending just the description of the changes.

**Reconciliation:** The function `reconcile` uses the update information computed in the previous step to build a set of transport instructions, each describing a path in one replica that is to be replaced by the corresponding subtree from the other replica.

**Propagation:** The function `propagate` actually performs the actions, taking the original filesystems `a` and `b` and the transport instructions `act` and returning the updated filesystems `a`$'$ and `b`$'$. The reason for performing reconciliation and propagation in two separate steps is that the full version of Unison actually inserts a user-interaction phase between the two, displaying the transport instructions and giving the user the opportunity to verify—and, if desired, override—Unison's recommendations before the changes are actually propagated.

These pieces are combined as follows to give the top-level synchronization function.

```
let sync o a b =
  let uia = buildUpdates o a in
  let uib = buildUpdates o b in
  let act = reconcile uia uib in
  let (a',b') = propagate o a b act in
  (a',b')
```

To give a more detailed picture of the job performed by each phase, it is useful to look at the data structures that flow between phases. (Indeed, these structures are very similar to those used in the actual Unison codebase.)

The inputs to the update detection phase of synchronization are the current replicas and the archive. All of these are filesystems, defined as follows. First, we define the abbreviations `name` and `contents` (both represented as `string`s) for the types of path names and file contents.

```
type name = string
type contents = string
```

Now, a filesystem  is either a directory (represented by the tag `Dir` plus a list of pairs of `name`s and subfilesystems), a file (represented by `File` plus a string), or nothing (represented by `Bot`). The `Bot` case is used to represent the complete absence of information—e.g., a missing archive or a completely unpopulated replica.

```
type fs =
    Dir of dContents
  | File of contents
  | Bot

and dContents = (name * fs) list
```

The buildUpdates function takes two filesystems—an archive and one of the replicas—and returns an updateItem representing the difference between the two. To define this type, we first define an enumerated type prevState describing what the filesystem previously contained at a given location.

```
type prevState =
    DIR
  | FILE
  | ABSENT
```

An updateItem has a tree structure corresponding to the modified part of the replica. Each subtree is either Same when the corresponding part of the filesystem are identical, Updated with an updateContent describing the difference between the two filesystems and a prevState describing the contents of the first filesystem, or Error, signalling that a non-fatal error happened during update detection (in the real implementation, this can be for instance an unreadable file). An updateContent may have different forms, depending on the new contents of the replica at this point. In case of a directory (UDir), it contains a list of pairs of name and updateItem corresponding to its children. In case of a file (UFile), it gives the new contents.

```
type updateItem =
    Same
  | Updated of updateContent * prevState
  | Error

and updateContent =
    UDir of updateChildren
  | UFile of contents
  | UAbsent

and updateChildren = (name * updateItem) list
```

The reconcile function takes two updateItems (one for each replica) and returns a transportInstrTree representing the actions that need to be performed to bring the two replicas into agreement (or as close as possible to full agreement). To define this type, we first define an enumerated type describing in what direction the propagation should take place. The directions Conflict and Equal correspond to a null action, but provide some information about the reason why no action should take place, which can be provided to the user.

```
type direction =
    Conflict
  | LeftToRight
  | RightToLeft
  | Equal
```

A transportInstrTree has a tree structure, indicating what location in the filesystem need to be modified. A transport instruction (of type transportInstr) is associated to each node of the tree. This can be an actual instruction Instr, a dummy instruction NoInstr in case nothing needs to be done at this location, or a dummy instruction Problem indicating that the synchronizer does not know what should be done at this location (because an error occured during update detection). Two update items (one for each replica) and a direction are associated to each instruction. The propagation agent uses one to check that the destination has not been modified since update detection and the other to actually propagate the changes.

```
type transportInstr =
    Instr of updateItem * updateItem * direction
  | NoInstr
  | Problem

type transportInstrTree =
       Node of transportInstr * transportInstrList
```

```
and transportInstrList =
      (name * transportInstrTree) list
```

We now informally sketch the correctness proof. To state the correctness of the implementation, we must first pick a particular instantiation of the definition of abstract filesystems—i.e., we must give the sets $\mathcal{S}$ of sorts and $\mathcal{N}$ of labels—and provide an abstraction function mapping our concrete filesystems to abstract ones.

For the bare-bones filesystems used by the present implementation, the only sorts we need are "directory," written DIR, and "file with contents *c*," written FILE(c).

$$\mathcal{S} = \{\text{DIR}\} \cup \{\text{FILE}(\texttt{c}) \mid \texttt{c} \in \texttt{contents}\}$$

We take the set of labels ($\mathcal{N}$) to be exactly the set of concrete names (names).

Define the auxiliary function *lookup*(ch, n) to yield the concrete filesystem associated with n in the association list ch, if there is one, and Bot otherwise. The abstraction function $\Uparrow \in \texttt{fs} \to \mathcal{F}$ is now defined as follows:

$$\begin{array}{lcl} \Uparrow(\texttt{Bot}) & = & \bot \\ \Uparrow(\texttt{Dir(ch)}) & = & \text{NODE}(\text{DIR}, \lambda\texttt{n}{\in}\texttt{names}.\ \Uparrow(\mathit{lookup}(\texttt{ch},\texttt{n})) \\ \Uparrow(\texttt{File(c)}) & = & \text{NODE}(\text{FILE}(\texttt{c}), \lambda\texttt{n}{\in}\texttt{names}.\ \bot). \end{array}$$

We say that an updateItem *accurately transforms* a filesystem fs into a filesystem fs′ if its prevState information is accurate and there is enough information to patch ⇑(fs) into ⇑(fs′).

Write $\texttt{t} \rhd \texttt{t}'$ to mean "evaluation of t yields result t′."

**7.1 Lemma:** If buildUpdates fs fs′ ▷ ui, then ui accurately transforms fs into fs′.

**Proof sketch:** By induction on fs, with a case analysis on the corresponding nodes of fs and fs′. □

If *f* is an abstract filesystem and act is an action tree, then *apply*(Left, act, *f*) denotes the filesystem obtained by applying the right-to-left updates described by act to *f*, while *apply*(Right, act, *f*) denotes the filesystem obtained by applying the left-to-right updates described by act to *f*.

We say that a transportInstrTree is *valid* for filesystems o, a and b if each updateItem accurately transforms o into the corresponding filesystem a or b.

**7.2 Lemma:** If reconcile uia uib ▷ act, and if uia and uib accurately transform the filesystem o into respectively a and b, then the run

$$\begin{array}{l}(\Uparrow(\texttt{o}),\ \Uparrow(\texttt{a}),\ \Uparrow(\texttt{b}),\\ \ \mathit{apply}(\texttt{Left},\texttt{act},\Uparrow(\texttt{o})),\ \mathit{apply}(\texttt{Right},\texttt{act},\Uparrow(\texttt{o})))\end{array}$$

is maximal and act is valid for o, a, and b.

**Proof sketch:** By induction on uib. □

**7.3 Lemma:** If

$$\texttt{propagate o a b act} \rhd (\texttt{a}',\texttt{b}'),$$

where act is valid for o, a, and b and the run

$$\begin{array}{l}(\Uparrow(\texttt{o}),\ \Uparrow(\texttt{a}),\ \Uparrow(\texttt{b}),\\ \ \mathit{apply}(\texttt{Left},\texttt{act},\Uparrow(\texttt{o})),\ \mathit{apply}(\texttt{Right},\texttt{act},\Uparrow(\texttt{o})))\end{array}$$

is maximal, then

$$(\Uparrow(\texttt{o}),\ \Uparrow(\texttt{a}),\ \Uparrow(\texttt{b}),\ \Uparrow(\texttt{a}'),\ \Uparrow(\texttt{b}'))) \text{ is maximal.}$$

**Proof sketch:** By induction on `act`. □

**7.4 Theorem:** If $\texttt{sync o a b} \rhd \texttt{(a}', \texttt{ b}')$, then $(\texttt{o}, \texttt{a}, \texttt{b}, \texttt{a}', \texttt{b}')$ is maximal.

**Proof:** By composing the above lemmas. □

(The above sketch assumes that no errors occur during synchronization. We have actually proved a stronger theorem for the full reference implementation, which also asserts that the run is safe even if some errors occur during either update detection or transport.)

The connection between the reference implementation and the actual Unison codebase is informal: the abstractions it embodies must be validated by means of engineering insight. However, the insight required is mostly quite local (as we discuss in the next section), so that the verification step, though manual, is *much* less demanding than checking that the full-blown Unison implementation meets its specification.

Moreover, once we believe the connection between the reference implementation and the real one, we can proceed from there on in a completely formal—indeed, machine-checked—manner. To demonstrate this, we have formalized the reference implementation as a set of inductive definitions in the language of the Coq proof assistant [2] and showed that it satisfies all the conditions in Sections 5 and 9; we also formalized the proofs of the properties of the specification in Section 6. This was a nontrivial but not oppressively difficult exercise in formalization and proof-script development, requiring a few weeks of effort by an experienced Coq "programmer." Besides increasing our confidence in the Unison implementation, this formalization exercise revealed a subtle bug in Unison's treatment of failures that had not previously been noticed (in some cases, a failure could be reported several times—not just for the file where it occurred, but also for some of the directories containing this file). In general, we felt that the effort we put into this formalization exercise was well repaid by its results.

# 8 The "Modeling Gap"

Fully specifying and verifying a real-world system program of Unison's size and complexity—i.e., proving that the actual codebase meets its specification—would be a superhuman task. (It would require, among other things, precise specifications of the Posix filesystem and socket APIs and the GTK user interface toolkit!) On the other hand, a program of Unison's complexity would be difficult to get right without a precise specification to guide the design. In building and rebuilding Unison over the years, we have tried to steer a middle way—working seriously on some days of the week on mathematical specifications and abstract reference implementations, writing code on other days, and living with the fact that there will always be a gap between one activity and the other.

Perhaps the most obvious difference between the real and reference implementations is that the reference implementation is a functional program—it takes the replicas and archive as arguments and returns new versions as a result—while the real one is imperative, modifying the real filesystems in-place.

In one sense, this difference is purely stylistic. The reference implementation never actually exploits the fact that, even after it "updates" a path in a filesystem, the old version of the filesystem is still available—i.e., filesystems are treated in a *linear* fashion—these "functional updates" might just as well be implemented as real, in-place updates. Indeed, a compiler with a sophisticated type analyzer might literally do this.

However, the difference between the functional and imperative formulations is correlated with a more problematic distinction: the reference implementation is written as if it "owns" the filesystems during synchronization, while the real implementation runs with live filesystems, where actions by users and other programs may modify the replicas while Unison is working.[5]

Unison addresses this discrepancy by noticing when concurrent modifications have taken place and signaling failures on those paths. It looks at each path in a given replica just *once* during update detection,

[5] This actually happens quite a bit in practice, since people often use Unison to manage multi-gigabyte replicas, for which update detection can take many minutes. Also, users often start Unison, leave it performing update detection, and then forget about it and do something else for a while before telling it to go ahead and propagate its changes.

extracts all the information it needs, and then never looks at the filesystem again until the moment when the contents of that path are being propagated between replicas in the final phase of synchronization. At this point, it double-checks, just before making any change to that path in a replica, that the contents of the path (in both replicas) are still the same as when it detected updates. Note, though, that this double-check is not completely reliable, since it is always possible that someone modifies the file just after the safety check but before the change is propagated; this is the cost of building Unison as portable, user-level program. (Also, note that this is not quite the same as taking a snapshot of the whole filesystem, since the update detection process will look at different files at different times; this difference does not seem to be confusing to users, though.)

Another related issue is that the real implementation operates on real file systems (using Posix system calls, file descriptors, inodes, etc.), whereas the reference implementation regards filesystems as simple, mathematical tree structures, which it inspects and manipulates directly. Most of the time, this difference does not cause serious confusion: in each place where the real implementation inspects or modifies a real filesystem, we can check (locally) that what it is doing corresponds to the action of the idealized implementation on its more abstract structure at the same point.

However, there are times when an operation that is easy to express in the reference implementation is very hard or even impossible to implement in terms of the operations provided by mainstream operating systems. For example, when Unison propagates a change to a file, it actually writes the new data into a temporary file as it comes off the network and, when the whole file has arrived, uses the `rename` system call to move it into place atomically. Unfortunately, this atomic `rename` only works under certain circumstances. On Windows systems, the new name must not refer to anything at the moment when `rename` is called (otherwise `rename` fails); on Unix, the new name may refer to an existing file (which is deleted as a side-effect of the `rename`), but not an existing directory. So, in general, before the new version of the file can be moved into place, the old file must be deleted or renamed to a backup version. If the host happens to crash between these two system calls, the replica will be left with *nothing* at this path, violating the specification. (Obviously, we could do better if we assumed a persistent storage substrate of the kind used to ensure durability of transactions in databases. Unfortunately, because Unison is designed to run as an ordinary application program, it can be "crashed" simply by killing its process, and there is no straightforward way to ensure that it will be executed again soon enough to process its transaction log and take appropriate action before the user has had a chance to observe the filesystem in its inconsistent state.)

A final difference between the reference implementation and the real one is that the reference implementation treats the archive as a full-blown filesystem, while the real implementation stores just a fingerprint of each file's contents. A consequence of the safety properties is that the contents of every file in a final filesystem $a'$ or $b'$ comes from either of the initial filesystems $a$ and $b$. So the synchronizer only needs the archive $o$ when checking for updates. This means that we do not actually need to keep the whole contents of $o$, as long as we retain enough information to tell which files have changed, and we can do this by keeping cryptographic fingerprints of the contents of each file, allowing us to detect modifications with high confidence.

# 9 Dealing with Metadata

As Unison matured and its user base grew, requests began to arrive for a variety of new features.[6] A large proportion of these had to do with handling various sorts of *metadata*—properties of files beyond just their contents, such as permissions, user-ids, group-ids, and modification times, as well as more esoteric properties such as the "spelling" of filenames on systems (such as Windows) where the capitalization of names is insignificant when reading and writing files but significant when listing contents of directories.

[6] One positive effect of working on the specification in parallel with the implementation was that it *slowed down* the process of engineering these features, making us reluctant to add a piece of functionality before we understood its effect on the specification—i.e., how it would behave in all cases and its full interactions will existing features. This process exposes a natural tension between users (who generally do not care at all how a given feature will behave in all cases, as long as it gives them the behavior they want in some case that is stopping them from getting work done at the moment) and developers (who have to deal with the long-term consequences of quick hacks).

There are two obvious ways to think about the metadata associated with a file or directory: we can either take metadata to be simply "part of the contents" of files, or we can consider metadata as separate and independent from contents. These lead to different views of how a synchronizer should propagate changes—in particular, it leads to different definitions of *conflicts.* If we take a file's metadata to be part of its contents, then, for example, changing a file's first byte on one replica and making it world-writable on the other replica should lead to a conflict—just as it would be a conflict if we changed a file's first byte on one replica and its second byte on the other. If we consider metadata as completely independent from contents, then in the above scenario we would expect the synchronizer to propagate the contents change in one direction and the permissions change in the other.

Unison actually chooses a middle way that, we feel, combines the virtues of these two extreme views, keeping the program's behavior easy to understand while avoiding some spurious conflicts. We regard a file's contents and its metadata as an *atomic* unit for purposes of propagating changes, but not for purposes of defining conflicts. That is, we require that each file's contents and all its metadata be equal, after synchronization, to the starting state of one of the replicas; however, if a part of the metadata is modified in an identical way in both replicas and the contents are modified just in one replica, no conflict is registered and the synchronizer is allowed to propagate the contents update.

The specification from Section 5 is easily refined to allow this behavior. We define an atomicity predicate *atomic* on sorts $\mathcal{S}$. Atomicity is extended to filesystems by saying that $f$ is atomic, written $\mathit{atomic}(f)$, if $f = \textsc{node}(s, \mathit{chld})$ and $s$ is atomic. We now add one new constraint to the definition of safety.

A run $(o, a, b, a', b')$ is said to *respect atomicity* (locally) if

$$\begin{array}{lcl} \mathit{atomic}(a') & \Longrightarrow & (a' = a) \vee (a' = b) \\ \mathit{atomic}(b') & \Longrightarrow & (b' = b) \vee (b' = a). \end{array}$$

Note that this property is always satisfied if the atomicity predicate is constantly false. So the original specification is indeed a special case of the refined one.

We instantiate the refined specification by introducing a set `properties` of file/directory metadata. The set $\mathcal{S}$ of sorts is defined as follows:

$$\begin{array}{rl} \mathcal{S} = & \{\textsc{dir},\ \textsc{file}\} \\ \cup & \{\textsc{filecontents}(\texttt{c}) \mid \texttt{c} \in \texttt{contents}\} \\ \cup & \{\textsc{fprops}(\texttt{pr}) \mid \texttt{pr} \in \texttt{properties}\} \\ \cup & \{\textsc{dprops}(\texttt{pr}) \mid \texttt{pr} \in \texttt{properties}\} \end{array}$$

We define the predicate *atomic* by $\mathit{atomic}(\textsc{file}) = \texttt{true}$ and $\mathit{atomic}(s) = \texttt{false}$ for all other sorts $s$. (This ensures that changes to the contents of a file and its associated meta-data will be treated atomically, while changes to directory metadata will be treated independently of changes to the directory's children.) Finally, we define the set of names $\mathcal{N}$ as the set of filenames plus two special labels, CONTENTS and PROPS:

$$\mathcal{N} = \texttt{names} \cup \{\textsc{contents},\ \textsc{props}\}.$$

This treatment of meta-data introduces one new issue into the modeling gap discussed in Section 8: the specification and reference implementation assume that file metadata such as modification times and permission bits are "semantically disjoint" from each other and from the file's contents, but in real filesystems this is sometimes not the case. Indeed, there are some cases where we cannot meet the specification at all because of the special properties of metadata. For example, if Unison finds it needs to create a file in a read-only directory (i.e., the directory on the other side was made writable, a file was created, and the directory was made read-only again), then the only way we can create the file is to make the directory briefly writable on this repository; if the program crashes just at this moment, the filesystem will be left in an incorrect state (with the directory properties not corresponding to the starting properties from either replica). The current version of Unison simply fails in this case (without making the directory writable on either side). Note that this is a failure to satisfy the specification—the result is not maximal—but a benign one—the result is still safe.

## 10 Modeling Failure

We close by sketching a last refinement of the reference implementation: the modeling of various sorts of failures during synchronization.

With all the possible sources of failure to which a synchronizer is subject (network problems, transient filesystem problems, user interruptions, concurrent updates during synchronization, bugs, etc.), failures are actually quite common in practice, and it is critical that a synchronizer behave safely and predictably when they occur. The Unison implementation goes to considerable effort to avoid misbehaving in case of failures, and we would like to be able to claim something about how well it succeeds. To this end, the full reference implementation and its formalized proof incorporate a model of run-time failures.

In the real Unison implementation, we distinguish two categories of failures. A *transient* failure (unreadable file, etc.) is local to one path and can be handled simply by skipping this path. Synchronization of other paths can be continued safely. *Fatal* errors, on the other hand, are either unexpected failures (synchronizer bugs, for instance) or disastrous conditions (user interruptions, permanent network problems, etc.). The only option in such cases is to display a diagnostic message and exit (leaving the filesystem and archive in a clean state!).

Our extension of the reference implementation models only transient errors, not fatal ones. It could be extended to handle fatal errors too, at some cost in readability. We have not done so because, in practice, fatal errors are simpler to think about. To protect against them, what we need to do is to make sure that, at every moment during execution, the state of the replicas (and the on-disk archive) satisfies the safety conditions in the specification (or that we have explicitly recorded the fact that we are briefly breaking them, as described in Section 8). This property of the implementation actually follows from our modeling of transient failures: at every moment, the state of the replicas must satisfy the safety conditions in the specification, because it is possible that all subsequent update operations may encounter transient failures. Transient errors, on the other hand, get incorporated into Unison's data structures and must be dealt with explicitly by "downstream" code.

The reference implementation is augmented to deal with failures by inserting explicit failure checks at every point where some operation is performed that might (in the real implementation) actually fail. We do not enforce the constraint that all such points are annotated with checks—this must be established by (straightforward) inspection of the code of the reference implementation.[7] Further details on the modeling of failures can be found at the beginning of Appendix A.

## 11 Future Work

We have not discussed Unison's cross-platform capabilities in this paper. In fact, a surprisingly large fraction of the engineering work on the implementation has gone into the tricky problems of synchronizing filesystems hosted on different operating systems—in particular, Posix and Win32 filesystems. We are experimenting with extending the specifications presented here to this setting. Some discussion of the engineering issues involved and how Unison addresses them can be found in [11]

Another important direction for further work is extending the specification ideas presented here to the case of multi-replica synchronization. In principle, Unison can actually be used for this purpose: $n$ replicas can be brought into agreement by performing $2n-3$ pairwise syncs. But, in practice, this method tends to become cumbersome for more than a small number of replicas. Moreover, the fact that Unison stores an archive per pair of replicas can lead to spurious conflicts if replicas are modified before all of the pairwise synchronizations have finished. Generalizing to multiple replicas thus demands a deeper re-examination of the basics of the specification; in particular, the archive should be replaced by a notion of "last common state"

[7] It is tempting to try to *enforce* failure checking at all points where it is required. To do this, we might try replacing the direct manipulation of the concrete tree datatype of filesystems by calls through an abstract "filesystem API" whose implementation would incorporate the required failure checks. However, this would make the correctness proofs *much* more complex. This is because it is only possible to write terminating functions in Coq. The termination of a function traversing a filesystem recursively is obvious if the filesystem is a concrete inductive data structure, but becomes much harder to see if the filesystem is treated abstractly.

derived from the causal history of the whole system. This step appears straightforward in outline, but involves some significant novelties compared to past work on causal histories and their realization in structures such as vector clocks. In particular, standard treatments of causal history do not consider "agreement events" where the synchronizer recognizes two equal replica states as synchronized, independent of their relation to the previous synchronized state—i.e., causally independent states are always considered to conflict. The consequences of adding such events to a standard account are not yet clear (mechanisms such as *version histories* [26, 9] and *hash histories* [12] are related, but these seem have been studied so far mainly from the perspective of algorithms for efficient implementation rather than of behavioral specifications).

A final direction concerns synchronization of structures other than filesystems. Our current project, dubbed Harmony `http://www.cis.upenn.edu/~bcpierce/harmony` , aims to extend the intuitions developed in Unison to synchronization of arbitrary tree-structured data—presented, for example, as XML documents [7, 22].

## Acknowledgements

Our main collaborator on the Unison system, Trevor Jim, also contributed many ideas to this specification. Sundar Balasubramaniam worked on an earlier synchronizer specification [1], which contributed many ideas to this one. Discussions with Peter Druschel, Sylvain Gommier, Matthieu Goulay, Michael Greenwald, Anne-Marie Kermarrec, Sanjeev Khanna, James Leifer, Norman Ramsey, Antony Rowstron, Marc Shapiro, Alan Schmitt, Zhe Yang, and participants on the `unison-users` mailing list have deepened our understanding of synchronization. Careful readings of earlier drafts by Jon Moore and Alan Schmitt helped us improve the presentation. Pierce's work on Unison was supported by the NSF under grants CCR-9701826 and ITR-0113226, *Principles and Practice of Synchronization*. Vouillon's was supported by a fellowship from the University of Pennsylvania's Institute for Research in Cognitive Science (IRCS).

# A Complete Reference Implementation

This appendix presents our reference synchronizer implementation in full, in the form of a complete, executable Objective Caml program.

The actual, machine-checked reference implementation is a collection of inductive definitions in Coq. We present it here in OCaml syntax, rather than Coq, as this is more likely to be familiar to readers. The OCaml code shown here was obtained by an automatic translation from the Coq definitions, followed by a little manual tidying for readability.

## A.1 Modeling Failure: The Action Monad

As we described in Section 10, the reference implementation models transient failures such as file-read errors by decorating each operation that might fail in the real implementation with a call to a "failure check" function.

The whole synchronizer is parameterized on a function `errors` from time (represented as an integer counter) to booleans.

```
errors : int -> bool
```

Checking whether an error occurs on a particular operation consists of applying the `errors` function to the current time and increasing the time by one. We can then say that "an error occurred during synchronization" if and only if the `error` function returns true for some time during synchronization.

Since the reference implementation is a functional program, the current time needs to be threaded through all function calls and returns. The notational burden of this threading is mitigated by encapsulating time within an abstract `action` type and reorganizing the code in a *monadic style* [33] so that time is passed along implicitly and automatically incremented every time `errors` is called. Specifically, we encapsulate time in an abstract `action` type and assume the existence of several functions that manipulate `action`s. First, we assume an evaluation function `eval` that performs an action: it takes the action and the starting time of the action and returns a pair containing the ending time of the action and the result of the evaluation of the action:

```
eval : 'a action -> int -> (int * 'a)
```

The action of testing for an error increments the time and returns a boolean indicating whether there was an error at the time when the action was performed. So we assume a function

```
test : bool action
```

which satisfies the equation

$$\texttt{eval test t} = \texttt{(t + 1, errors t)}. \qquad (1)$$

There are two other ways of creating an action. We can define the action of evaluating an expression using the operator below.

```
return : 'a -> 'a action
```

such that

$$\texttt{(eval (return e) t)} = \texttt{(t, e)}. \qquad (2)$$

Two actions can be performed in sequence using the `bind` operator

```
bind : 'a action -> ('a -> 'b action) -> 'b action
```

which satisfies the property that,

$$\begin{array}{ll} \text{if} & \texttt{(eval e t)} = \texttt{(t}', \texttt{ v)} \\ & \texttt{(eval (f v) t}'\texttt{)} = \texttt{(t}'', \texttt{ v}'\texttt{)} \\ \text{then} & \texttt{(eval (bind e f) t)} = \texttt{(t}'', \texttt{ v}'\texttt{)}. \end{array} \tag{3}$$

The `bind` operator can be understood intuitively as a `let`-binding: the expression `bind e (fun x -> e′)` corresponds to `let x = e in e′` but keeps track of the time.

In the proof of correctness of the reference implementation, the concrete definitions of the functions `bind`, `test`, `return`, and `eval` are left unspecified; the proof relies only on the axioms (1)–(3) above. This ensures that the proof guarantees safety under *arbitrary* failure scenarios.

Of course if we want to actually experiment with the reference implementation, we must supply definitions of these functions. Here, for example, is a trivial implementation that never generates any errors:

```
type 'a action = 'a
let bind (x: 'a action) (f: 'a -> 'b action) = f x
let test : 'a action = (t, false)
let return (x: 'a) : 'a action = x
let eval (x: 'a action) (t: int) = t * x
```

Here is a more interesting implementation that generates a transient error on every fifth operation:

```
type 'a action = int -> (int * 'a)
let bind (x: 'a action) (f: 'a -> 'b action) =
  fun (t: int) ->
    let (t',r) = x t in
    f r t'
let test : bool action = fun (t: int) -> (t, t mod 5 = 0)
let return (x: 'a) : 'a action = fun (t: int) -> (t,x)
let eval (x: 'a action) (t: int) = x t
```

## A.2 Filesystems

The inputs to the update detection phase of synchronization are the current replicas and the archive. All of these are filesystems, defined as follows.

First, we define the abbreviations `name`, `contents` and `properties` (all represented as `string`s) for the types of path names, file contents and file properties.

```
type name = string
type contents = string
type properties = string
```

Now, a filesystem  is either a directory (represented by the tag `Dir` plus a pair of properties and a list of pairs of `name`s and sub-filesystems), a file (represented by `File` plus a pair of properties and contents), a symbolic link (represented by `Symlink` plus a contents) or nothing (represented by `Bot`).[8]

```
type fs =
    Dir of properties * dContents
  | File of properties * contents
  | Symlink of contents
  | Bot
and dContents = (name * fs) list
```

[8] At this point, the implementation does not precisely follow the general formulation proposed in Section 9. Instead, file and directory properties are built directly into `File` and `Directory` nodes in the filesystem. We have not done so, but this formulation could be related to the general one by means of an abstraction function.

Formally, we do not bother ensuring that dContents is a set; instead, we assume that it may contain duplicate bindings for a single name, but that all bindings except the first one are ignored.

Some utility functions on filesystems will be useful later. The expression assoc n c returns the filesystem associated to the name n in the directory contents c.

```
let rec assoc n c =
  match c with
    []              -> Bot
  | (n', fs) :: rem -> if n = n' then fs else assoc n rem
```

The function lookupPath returns the filesystem contained at path p in the filesystem f.

```
let rec lookupPath p f =
  match p with
    [] -> f
  | n :: p' ->
      assoc n
        (match lookupPath p' f with
           Dir (pr, c) -> c
         | _           -> [])
```

The function reverse reverses a path (actually it is just the standard polymorphic list reverse):

```
let rec reverse p1 p2 =
  match p1 with
    []       -> p2
  | n :: p1' -> reverse p1' (n :: p2)
```

The function mem checks whether a child named n is among the list of (name, filesystem) pairs l:

```
let rec mem n l =
  match l with
    []              -> false
  | (n', fs) :: rem -> n = n' || mem n rem
```

## A.3 Update Detection

The buildUpdates function takes two filesystems—an archive and one of the replicas—and returns an updateItem representing the difference between the two. To define this type, we first define an enumerated type prevState describing what the filesystem previously contained at a given location.

```
type prevState =
    DIR
  | FILE
  | SYMLINK
  | ABSENT
```

The function fsKind associate to a filesystem the state of its root.

```
let fsKind = function
  Dir _     -> DIR
| File _    -> FILE
| Symlink _ -> SYMLINK
| Bot       -> ABSENT
```

A leafUpdate records a modification at a leaf of the replica: a property or content modification. It can be either LeafSame in case the corresponding part of filesystem are identical, or LeafUpdated with a pair of a new value and an old value (if there was any) in case of a modification.

```
type 'a leafUpdate =
    LeafSame
  | LeafUpdated of 'a * 'a option
```

An `updateItem` has a tree structure corresponding to the modified part of the replica. Each subtree is either `Same` when the corresponding part of the filesystem are identical, `Updated` with an `updateContent` describing the difference between the two filesystems and a `prevState` describing the contents of the first filesystem, or `Error`, signalling that a non-fatal error happened during update detection (in the real implementation, this can be for instance an unreadable file). An `updateContent` may have different forms, depending on the new contents of the replica at this point. In case of a directory (`UDir`), it contains a `leafUpdate` for its properties and a list of pairs of `name` and `updateItem` corresponding to its children. In case of a file (`UFile`), it contain the properties and contents `leafUpdate`s. In case of a symlink, it contain a contents `leafUpdate`.

```
type updateItem =
    Same
  | Updated of updateContent * prevState
  | Error
and updateContent =
    UCDir of properties leafUpdate * updateChildren
  | UCFile of properties leafUpdate * contents leafUpdate
  | UCSymlink of contents leafUpdate
  | UCAbsent
and updateChildren = (name * updateItem) list
```

To define the required operations on `updateItem`s, we first need a couple of utility functions. The first is a generic function `remove`, which takes as argument a key `n` and a list `l` of (key,value) bindings, and returns a list `l` where all bindings with key `n` have been removed.

```
let rec remove n l =
  match l with
    [] ->                []
  | (n', ui) :: rem -> if n = n' then remove n rem else (n', ui) :: remove n rem
```

Another utility function, `deletions`, generates a list of `updateChildren` representing the fact that all the files in the list `c` have been deleted.

```
let rec deletions c =
  match c with
    []               -> []
  | (n, Bot) :: rem  -> remove n (deletions rem)
  | (n, fs) :: rem   -> (n, Updated (UCAbsent, fsKind fs)) :: deletions rem
```

The main update detection function, `buildUpdates`, takes an `archive` and a filesystem `fs` and returns an appropriate `updateItem`. Its first action is to check for an error and return the special `updateItem` `Error` if so. Otherwise, it considers that the root of the filesystem `fs` can be accessed and builds an `updateItem` by pattern matching on the state of the filesystem and of the archive `archive`. In the case where the current filesystem contains a directory, it uses `buildUpdateChildren` to recursively process the contents of the directory.

```
let rec buildUpdates archive fs =
  bind test (fun err ->
  if err then return Error else
  match (archive, fs) with
    (Dir (pr0, c0), Dir (pr, c)) ->
      bind (buildUpdateChildren c0 c) (fun uch ->
      let up = if pr0 = pr then LeafSame else LeafUpdated (pr, Some pr0) in
      if uch = [] && up = LeafSame then return Same
      else return (Updated (UCDir (up, uch), DIR)))
  | (_, Dir (pr, c)) ->
      bind (buildUpdateChildren [] c) (fun uch ->
      return (Updated (UCDir (LeafUpdated (pr, None), uch), fsKind archive)))
  | (File (pr0, c0), File (pr, c)) ->
      let up = if pr0 = pr then LeafSame else LeafUpdated (pr, Some pr0) in
      let uc = if c0 = c then LeafSame else LeafUpdated (c, Some c0) in
      if up = LeafSame && uc = LeafSame then return Same
      else return (Updated (UCFile (up, uc), FILE))
  | (_, File (pr, c)) ->
      return (Updated (UCFile (LeafUpdated (pr, None),
                               LeafUpdated (c, None)),
                       fsKind archive))
  | (Symlink c0, Symlink c) ->
      if c0 = c then return Same
      else return (Updated (UCSymlink (LeafUpdated (c, Some c0)),
                            fsKind archive))
  | (_, Symlink c) ->
      return (Updated (UCSymlink (LeafUpdated (c, None)), fsKind archive))
  | (Bot, Bot) ->
      return Same
  | (_, Bot) ->
      return (Updated (UCAbsent, fsKind archive)))

and buildUpdateChildren c' c =
  match c with
    [] ->
      return (deletions c')
  | (n, fs) :: rem ->
      bind (buildUpdates (assoc n c') fs) (fun ch ->
      bind (buildUpdateChildren c' rem) (fun uc ->
      let rem' = remove n uc in
      if ch = Same then return rem' else return ((n, ch) :: rem')))
```

Note that `buildUpdates` yields a *minimal* `updateItem` tree, mentioning the paths where updates have occurred. For instance, a given directory will be mentioned in the tree only if either its properties have changed or one of its children has been updated. It is important to keep the `updateItem` tree small because, in the actual implementation, the `updateItem` for one of the replicas must be transmitted over the network in order to be compared with the other during the reconciliation step.

## A.4 Reconciliation

The `reconcile` function takes two `updateItem`s, one for each replica, and returns a `transportInstrTree` representing the actions that need to be performed to bring the two replicas into agreement (or, if there are conflicts or errors, as close as possible to full agreement). To define this type, we first define an enumerated type describing in what direction the propagation should take place.

```
type direction =
    Conflict
  | LeftToRight
  | RightToLeft
  | Equal
```

The directions `Conflict` and `Equal` both direct the next phase (transport) not to take any action for a particular path, but the distinction provides some information about the reason why no action should take place, which (in the actual implementation) can be displayed to the user.

A transportInstrTree has a tree structure, indicating what location in the filesystem need to be modified. A transport instruction (of type transportInstr) is associated to each node of the tree. This can be an actual instruction Instr, a dummy instruction NoInstr in case nothing needs to be done at this location, or a dummy instruction Problem indicating that an error occured during update detection. Two update items (one for each replica) and a direction are associated to each instruction. The propagation agent uses one to check that the destination has not been modified since update detection and the other to actually propagate the changes.

```
type transportInstr =
    Instr of updateItem * updateItem * direction
  | NoInstr
  | Problem

type transportInstrTree = Node of transportInstr * transportInstrList
and transportInstrList = (name * transportInstrTree) list
```

The propagateErrors function takes a transport instruction act as argument and checks whether any of the update items associated with the instruction contain an Error. If there is no Error, the transport instruction is returned as-is; otherwise, Problem is returned. The hasErrors function checks whether an update item ui contains an error.

```
let rec hasErrors ui =
  match ui with
    Updated (UCDir (_, uch), _) -> hasErrorsChildren uch
  | Error                       -> true
  | _                           -> false

and hasErrorsChildren uch =
  match uch with
    [] -> false
  | (n, ui) :: rem -> hasErrors ui || hasErrorsChildren rem

let propagateErrors act =
  match act with
    Instr (ui, ui', d) ->
      if hasErrors ui || hasErrors ui' then Problem else act
  | _ ->
      act
```

The reconcileNoConflict function handles the case when one of the replicas has not been changed—i.e., where what needs to be done is just to copy its current state onto the other replica. The argument s of type side indicates which is the source replica.

```
type side =
    Left
  | Right

let noConflictInstr s ui =
  match s with
    Left  -> Instr (ui, Same, LeftToRight)
  | Right -> Instr (Same, ui, RightToLeft)

let rec reconcileNoConflict s ui =
  match ui with
    Same -> Node (NoInstr, [])
  | Updated (UCDir (up, uch), DIR) ->
      let act =
        match up with
          LeafSame      -> NoInstr
        | LeafUpdated _ -> noConflictInstr s ui
      in
      Node (act, reconcileNoConflictChildren s uch)
  | Updated _ ->
      Node (propagateErrors (noConflictInstr s ui), [])
  | Error ->
      Node (Problem, [])

and reconcileNoConflictChildren s uc =
  match uc with
    [] -> []
  | (n, ui) :: rem ->
      (n, reconcileNoConflict s ui) :: reconcileNoConflictChildren s rem
```

Note that the propagateErrors function is not called for instructions corresponding to a locally updated directory. In this case, only the property part of the updateItem will be used.

The leafDirection and combineDirections functions are used to generate instructions for leaf updates. The leafDirection function decides in which direction the update should take place. The combineDirections function computes the direction corresponding to two distinct updates (a file content update and a file property update).

```
let leafDirection u u' =
  match (u, u') with
    (LeafUpdated (v, _), LeafUpdated (v', _)) ->
       if v = v' then Equal else Conflict
  | (LeafSame, LeafSame) ->
       Equal
  | (LeafUpdated _, LeafSame) ->
       LeftToRight
  | (LeafSame, LeafUpdated _) ->
       RightToLeft

let combineDirections d d' =
  match d, d' with
    (Conflict, _) | (_, Conflict)
  | (LeftToRight, RightToLeft) | (RightToLeft, LeftToRight) ->
      Conflict
  | (LeftToRight, _) | (_, LeftToRight) ->
      LeftToRight
  | (_, RightToLeft) | (RightToLeft, _) ->
      RightToLeft
  | (Equal, Equal) ->
      Equal
```

The utility function uassoc looks up updateItems in association lists: uassoc n c returns the updateItem associated to the name n in the list of children c, or Same if none exists.

```
let rec uassoc n uch =
  match uch with
    []              -> Same
  | (n', ui) :: rem -> if n = n' then ui else uassoc n rem
```

The top-level function in the reconciliation phase is reconcile; it takes two update items representing the updates that have occurred on the two replicas with respect to the last synchronized state, and returns a transportInstrTree structure that can be applied to both replicas (as explained in the following subsection) to yield new replicas that are as close to synchronized as possible. It proceeds by cases on the top nodes of the two updateItems: if one is Same, then it uses reconcileNoConflict to construct a transportInstrTree that simply copies state of the other replica on top of the one that has not changed; if both are updated directories, then it calls reconcileChildren to (recursively) deal with the contents; if both are files, it uses combineDirections to appropriately (i.e., atomically) combine the changes to the contents and properties; otherwise it signals a conflict. The function reconcileChildren first handles the directory contents that are modified on the second replica, then calls the helper function reconcileLeft to handle the contents which are modified only on the first ("left") replica.

```
let rec reconcile uia uib =
  match (uia, uib) with
    (_, Error) | (Error, _) ->
      Node (Problem, [])
  | (_, Same) ->
      reconcileNoConflict Left uia
  | (Same, _) ->
      reconcileNoConflict Right uib
  | (Updated (UCDir (upa, ucha), _), Updated (UCDir (upb, uchb), _)) ->
      let act =
        if upa = LeafSame && upb = LeafSame then NoInstr
        else Instr (uia, uib, leafDirection upa upb) in
      Node (act, reconcileChildren uchb ucha uchb)
  | (Updated (UCFile (upa, uca), _), Updated (UCFile (upb, ucb), _)) ->
      Node (Instr (uia, uib,
                   (combineDirections (leafDirection uca ucb)
                                      (leafDirection upa upb))),
            [])
  | (Updated (UCSymlink uca, _), Updated (UCSymlink ucb, _)) ->
      Node (Instr (uia, uib, leafDirection uca ucb), [])
  | _ ->
      Node (propagateErrors (Instr (uia, uib, Conflict)), [])

and reconcileChildren uchb0 ucha uchb =
  match uchb with
    [] ->
      reconcileLeft uchb0 ucha
  | (n, ui) :: rem ->
      (n, reconcile (uassoc n ucha) ui) :: reconcileChildren uchb0 ucha rem

and reconcileLeft uchb ucha =
  match ucha with
    [] ->  []
  | (n, ui) :: rem ->
      match uassoc n uchb with
        Same -> (n, reconcileNoConflict Left ui) :: reconcileLeft uchb rem
      | _    -> reconcileLeft uchb rem
```

## A.5 Propagation

The final phase, propagation, takes the transportInstrTree calculated by the reconciler and applies it to the current replicas.

Because of the possibility of concurrent updates to the replicas during synchronization, the real implementation goes to considerable trouble to be as paranoid as possible when propagating updates. Instead of blindly overwriting the replica with the new contents, it first transfers the new information to a temporary file (or directory structure) on the same host, then double-checks that the current contents of that replica are exactly the same as when update detection was performed, and only then moves the new information into its final position. This double-checking process is modeled in the reference implementation by providing *five* inputs to the top-level synchronization function (see Section A.6): the archive o, the two replicas a and b, and two more filesystems a′ and b′, representing the states of the replicas just at the moment when propagation begins. The propagation code in this section uses o plus the update items for each of the replicas to construct two “predicted current replicas” and compares these against the current replicas a′ and b′.

The function updateArchive takes an archive o and an update item ui and computes the expected contents of one of the replicas, starting from the archive and an update item corresponding to what has been modified in the replica compared to the archive contents. The helper function leafApply performs an update u at a leaf (a file contents or properties) value v.

```
let leafApply v u =
  match u with
    LeafSame            -> v
  | LeafUpdated (v’, _) -> v’

let rec updateArchive o ui =
  match (ui, o) with
    (Updated (UCAbsent, _), _) ->
       Bot
  | (Updated (UCFile (up, uc), _), File (pr, c)) ->
       File (leafApply pr up, leafApply c uc)
  | (Updated (UCFile (LeafUpdated (pr, _), LeafUpdated (c, _)), _), _) ->
       File (pr, c)
  | (Updated (UCSymlink (LeafUpdated (c, _)), _), _) ->
       Symlink c
  | (Updated (UCDir (up, uch), _), Dir (pr, ch)) ->
       Dir (leafApply pr up, updateArchiveChildren ch uch)
  | (Updated (UCDir (LeafUpdated (pr, _), uch), _), _) ->
       Dir (pr, updateArchiveChildren [] uch)
  |  _ ->
       o

and updateArchiveChildren ch uch =
  match uch with
    [] ->
      ch
  | (n, ui) :: ucs’ ->
      let f = updateArchive (assoc n ch) ui in
      let ch’ = remove n (updateArchiveChildren ch ucs’) in
      if f = Bot then ch’ else (n, f) :: ch’
```

The function emptySource checks that the source filesystem f contains nothing at path p, and that the filesystem at path p in a is immediately contained in a directory. Note that we have a conflict if the latter condition is not satisfied.

```
let emptySource p f =
  (match p with
     []      -> true
   | n :: p’ -> match lookupPath p’ f with (Dir _) -> true | _ -> false)
  &&
  (lookupPath p f = Bot)
```

The next helper function reads out a “copy” of an archive filesystem o from a current filesystem f. It returns None if f cannot be found inside o. (The reason we take the trouble to perform this copy, rather than just using o directly, is that, in the real implementation, the archive is stored in compressed form, using cryptographic fingerprints in place of full file contents. The job of copy is, using f, to reconstruct the full filesystem corresponding to o.)

```
let rec copyRec o f =
  match (o, f) with
    (Dir (pro, cho), Dir (pr, ch)) ->
      if pro = pr then
        match copyChildren ch [] cho with
          Some ch' -> Some (Dir (pr, ch'))
        | None     -> None
      else
        None
  | (File (pro, co), File (pr, c)) ->
      if pro = pr && co = c then Some f else None
  | (Symlink co, Symlink c) ->
      if co = c then Some f else None
  | (Bot, _) ->
      Some Bot
  | _ ->
      None

and copyChildren ch ch0 cho =
  match cho with
    [] ->
      Some []
  | (n, o) :: cho' ->
      if mem n ch0 then copyChildren ch ch0 cho' else
      match copyRec o (assoc n ch), copyChildren ch ((n, o) :: ch0) cho' with
        Some f, Some ch' -> Some ((n, f) :: ch')
      | _                -> None

let copy p o f =
  bind test (fun err ->
  if err then return None else
  match o with
    Bot -> return (if emptySource p f then Some Bot else None)
  | _   -> return (copyRec o (lookupPath p f)))
```

We use the function checkNoUpdates just before we make each change to a replica, to verify (as a sanity check) that the current contents matches the previous contents recorded in the transport instruction.

```
let checkNoUpdates o f p ui =
  bind (buildUpdates (updateArchive (lookupPath p o) ui) (lookupPath p f))
    (fun updates ->
  return (updates = Same))
```

The replace function returns a new version of an input filesystem f in which the contents at a path p has been replaced by another filesystem g. The filesystem f is returned unchanged if there is an error or if the parent directory of the endpoint of the path p does not exist. The inner loop of this function, replaceRec, performs no additional checking for transient errors: we assume that, if an error is going to happen during replacement, it happens at the beginning. (In other words, the replacement is assumed to be atomic.)

```
let rec replaceRec p f g =
  match p, f with
    [], _                -> g
  | n :: p', Dir (pr, ch) -> Dir (pr, (n, replaceRec p' (assoc n ch) g) :: remove n ch)
  | _                    -> f
```

```
let replace p f g =
  bind test (fun err ->
  if err then return f else return (replaceRec (reverse p []) f g))
```

(The path p needs to be reversed because in replaceRec we simultaneously traverse p and f starting at the root; p starts out in the wrong order for this traversal.)

The function performInstrLeaf is a bulletproofed wrapper for replace. The function performInstrLeaf performs a bulletproofed replacement of the target contents by the source contents (as specified by o).

```
let performInstrLeaf p o src trg usrc utrg =
  bind (copy p (updateArchive (lookupPath p o) usrc) src) (fun cpy ->
  match cpy with
    Some o'' ->
      bind (checkNoUpdates o trg p utrg) (fun noUpdates ->
      if noUpdates then replace p trg o'' else return trg)
  | None ->
      return trg)
```

The function newProps calculates the new properties (if necessary) for a given directory.

```
let newProps o ui =
  match (ui, o) with
    (Updated (UCDir (up, _), _), Dir (pr, ch)) -> Some (leafApply pr up)
  | (Same,  (Dir (pr, _)))                     -> Some pr
  | _                                          -> None
```

The unchangedProps function tests whether the properties in a given directory have changed since we recorded them during update detection.

```
let unchangedProps pr f =
  match f with
    Dir (pr', ch) -> pr = pr'
  | _             -> false
```

The function performInstrDir is called to update just the properties of an existing directory. It takes the same arguments as performInstrLeaf, and returns a new version of the target in which the properties for the directory at path p have been copied from the source. If the contents of the target are not a directory or if the properties of either the source or the target have changed since we looked at them originally, then we return the target unchanged.

```
let performInstrDir p o src trg usrc utrg =
  bind test (fun err ->
  if err then return trg else
  match newProps (lookupPath p o) usrc, newProps (lookupPath p o) utrg with
    Some prsrc, Some prtrg ->
      if unchangedProps prsrc (lookupPath p src) && unchangedProps prtrg (lookupPath p trg)
      then match lookupPath p trg with
             Dir (_, ch) -> replace p trg (Dir (prsrc, ch))
           | _           -> return trg
      else return trg
  | _ ->
      return trg)
```

The function performInstr dispatches to performInstrDir or performInstrLeaf, depending on the form of the update item it is processing.

```
let performInstr p o src trg usrc utrg =
  match usrc with
    Updated (UCDir _, _) -> performInstrDir p o src trg usrc utrg
  | _                    -> performInstrLeaf p o src trg usrc utrg
```

The main workhorse function for update propagation is propagateLocally. Given a path p, an archive o, current replicas $a'$ and $b'$ (possibly not identical to the a and b on which the update items were originally computed) and a transport instruction act, it destructs act to see what needs to be done and then either returns $(a',b')$ unchanged (if the instruction indicates a conflict or already-equal replicas) or calls performInstr to actually perform the action. Note that performInstr assumes that changes are to be propagated from left to right; to propagate from right to left, we exchange the senses of a and b when we call it.

```
let propagateLocally p o a' b' act =
  match act with
    Instr (ua, ub, dir) ->
      begin match dir with
        Conflict ->
          return (a', b')
      | LeftToRight ->
          bind (performInstr p o a' b' ua ub) (fun b'' ->
          return (a', b''))
      | RightToLeft ->
          bind (performInstr p o b' a' ub ua) (fun a'' ->
          return (a'', b'))
      | Equal ->
          return (a', b')
      end
  | _ ->
      return (a', b')
```

The main function propagate recursively traverses the transportInstr tree and executes the corresponding instructions. The helper function propagateInChildren (which recursively calls propagate) is used to apply actions at paths strictly longer than p; then propagateLocally is used to propagate the action at p itself.

```
let rec propagate p o a b = function
  Node (act, ts) ->
    bind (propagateInChildren p o a b ts) (fun fs ->
    let (a', b') = fs in
    propagateLocally p o a' b' act)

and propagateInChildren p o a b t =
  match t with
    [] ->
      return (a, b)
  | (n, act) :: acts' ->
      bind (propagate (n :: p) o a b act) (fun fs ->
      let (a', b') = fs in
      propagateInChildren p o a' b' acts')
```

## A.6 Main Program

Finally, sync binds all of the phases described above into a complete synchronization function. Note that it takes extra inputs $a'$ and $b'$ and passes these, rather than a and b, to propagate, modeling the possibility of concurrent changes to the replicas during update detection and reconciliation.

```
let sync o a b a' b' =
  bind (buildUpdates o a) (fun ua ->
  bind (buildUpdates o b) (fun ub ->
  propagate [] o a' b' (reconcile ua ub)))
```

## References

[1] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, Oct. 1998. Full version available as Indiana University CSCI technical report #507, April 1998.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.

[3] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, Apr. 1982.

[4] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3), September 1985.

[5] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California*, December 1994.

[6] W. K. Edwards, E. D. Mynatt, K. Petersen, M. Spreitzer, D. B. Terry, and M. Theimer. Designing and implementing asynchronous collaborative applications with bayou. In *ACM Symposium on User Interface Software and Technology*, pages 119–128, 1997.

[7] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2004. Long version available as University of Pennsylvania technical report MS-CIS-03-08.

[8] R. G. Guy, G. J. Popek, and T. W. P. Jr. Consistency algorithms for optimisic replication. In *Proceedings of the First International Conference on Network Protocols*, October 1993.

[9] J. H. Howard. Reconcile user's guide. Technical Report TR99-14, Mitsubishi Electronics Research Lab, 1999.

[10] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the 8th European Conference on Computer-Supported Cooperative Work*, September 2003. Helsinki, Finland.

[11] T. Jim, B. C. Pierce, and J. Vouillon. How to build a file synchronizer. Manuscript; available from `http://www.cis.upenn.edu/~bcpierce/papers`, 2001.

[12] B. B. Kang, R. Wilensky, and J. Kubiatowicz. Hash history approach for reconciling mutual inconsistency in optimistic replication. In *23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, 2003.

[13] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Principles of Distributed Computing (PODC)*, 2001.

[14] J. J. Kistler. *Disconnected Operation in a Distributed File System.* PhD thesis, Carnegie Mellon University, 1996.

[15] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX Winter*, pages 95–106, 1995.

[16] X. Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

[17] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. Safe generic data synchronizer. Rapport de recherche, LORIA France, May 2003.

[18] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proceedings of ACM Group 2003 Conference*, November 9–12 2003. Sanibel Island, Florida.

[19] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 11(1), December 1997.

[20] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, October 1997.

[21] S. H. Phatak and B. Badrinath. Conflict resolution and reconciliation in disconnected databases. In *Mobility in Databases and Distributed Systems (MDDS), Florence, Italy*, Sept. 1999.

[22] B. C. Pierce, A. Schmitt, and M. B. Greenwald. Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, 2003.

[23] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. In *Proceedings of the 8th European Software Engineering Conference*, pages 175–185. ACM Press, 2001.

[24] P. Reiher. Rumor 1.0 User's Manual., 1997. http://fmg-www.cs.ucla.edu/rumor.

[25] P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the ficus file system. In *USENIX Summer Conference Proceedings*, pages 183–195, 1994.

[26] B. Richard, D. M. Nioclais, and D. Chalon. Clique: a transparent, peer-to-peer collaborative file sharing system. In *Proceedings of the 4th international conference on mobile data management (MDM '03)*, Jan. 21-24 2003. Melbourne, Australia.

[27] Y. Saito and M. Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, HP Laboratories Palo Alto, Feb. 8 2002.

[28] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file systems for a distributed workstation environment. *IEEE Transactions on Computers*, C-39(4):447–459, Apr. 1990.

[29] M. Shapiro, A. Rowstron, and A.-M. Kermarrec. Application-independent reconciliation for nomadic applications. In *Proc. SIGOPS European Workshop: "Beyond the PC: New Challenges for the Operating System"*, Kolding (Denmark), Sept. 2000. ACM SIGOPS. `http://www-sor.inria.fr/~shapiro/papers/ew2000-logmerge.html`.

[30] SyncML: The new era in data synchronization. `http://www.syncml.org`.

[31] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30, 1994*, pages 140–149. IEEE Computer Society, 1994.

[32] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado*, December 1995.

[33] P. Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences.* Springer-Verlag, Aug. 1992. Also in J. Jeuring and E. Meijer, editors, Advanced Functional Programming, Springer Verlag, LNCS 925, 1995.