Schema-Directed Data Synchronization

J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard,
Benjamin C. Pierce, and Alan Schmitt

Technical Report MS-CIS-05-02
Department of Computer and Information Science
University of Pennsylvania

Supersedes MS-CIS-03-42

March 23, 2005

Abstract

Increased reliance on optimistic data replication has led to burgeoning interest in tools and frameworks for *synchronizing* disconnected updates to replicated data. We have implemented a generic, synchronization framework, called Harmony, that can be instantiated to yield state-based synchronizers for a wide variety of tree-structured data formats. A novel feature of this framework is that the synchronization process—in particular, the recognition of situations where changes are in conflict—is driven by the schema of the structures being synchronized.

We formalize Harmony's synchronization algorithm, prove that it obeys a simple and intuitive specification, and illustrate how it can be used to synchronize a variety of specific forms of application data—sets, records, tuples, and relations.

1 Introduction

Optimistic replication strategies are attractive in a growing range of settings where weak consistency guarantees can be accepted in return for higher availability and the ability to update data while disconnected. These uncoordinated updates must later be *synchronized* (or *reconciled*) by automatically combining non-conflicting updates while detecting and reporting conflicting updates.

Our long-term aim is to develop a generic framework that can be used to build high-quality synchronizers for a wide variety of application data formats with minimal effort. As a step toward this goal, we have designed and built a prototype synchronization framework called Harmony, focusing on the important special cases of unordered and rigidly ordered data (including sets, relations, tuples, records, feature trees, etc.) and offering only limited support for list-structured data such as structured documents. An instance of Harmony that synchronizes multiple calendar formats (Palm Datebook, Unix ical, and iCalendar) is in daily use within our group; we are also developing Harmony instances for bookmark data (handling the formats used by several common browsers, including Mozilla, Safari, OmniWeb, Internet Explorer 5, and Camino), address books, application preference files, drawings, and bibliographic databases.

The Harmony system embodies two major novelties: (1) A generic, *schema-directed* synchronization algorithm, whose behavior is controlled by the schema of the structures being synchronized, and (2) a domain-specific programming language for writing bi-directional transformations on trees, which we use to convert heterogeneous concrete data formats into a common format for synchronization. The latter has been described elsewhere [12]; the former is the topic of the present paper.

The primary importance of schema information in Harmony's synchronization algorithm is in detecting conflicts—recognizing situations where changes in one replica may *not* be propagated to the other because the resulting combined structure would be ill-formed. To our knowledge, Harmony is the first state-based synchronizer to preserve structural invariants on richer objects than raw untyped trees.[1]

The intuition behind the algorithm is quite simple: we try to propagate changes from each replica to the other, validate the resulting trees according to the expected schema, and signal a conflict if validation fails. However, this is not as trivial as it may sound: there may be many changes to propagate from each replica to the others, leading to many possible choices of *where* to signal conflicts (i.e., which subset of the changes are actually propagated). To ensure progress, we want synchronization to propagate as many changes as possible while respecting the schema; at the same time, to avoid surprising users, we need the results of synchronization to be stable, in the sense that small variations in the inputs cannot produce large variations in the set of changes that are propagated. A natural way of combining these design constraints is to demand that the results of synchronization be *maximal*, in the sense that, if there is *any* well-formed way to propagate a given change from one replica to the other that does not violate schema constraints, then that change *must* be propagated.

Our main technical contribution is a proof that, for schemas satisfying a locality constraint called *path consistency* (a semantic variant of the *consistent element declaration* condition in W3C Schema), a simple one-pass, recursive tree-walking algorithm does indeed yield results that are maximal in this sense.

After establishing some notation in Section 2, we warm up in Section 3 with some simple synchronization examples. Section 4 analyzes several forms of conflict, in particular introducing the notion of *schema conflict*, which plays a crucial role in the synchronization algorithm. Section 5 presents the algorithm and its formal specification (in particular, the definition of maximality) and proves that the two correspond. Section 6 illustrates the behavior of the algorithm on more realistic schemas. Sections 7 and 8 discuss related and future work.

2 Notation

Internally, Harmony manipulates structured data in an extremely simple form: unordered, edge-labeled trees. Richer external formats such as XML are encoded as unordered trees. We chose this simple data model on pragmatic grounds: experience has shown that the reduction in the overall complexity of the Harmony system far outweighs the complexity due to manipulating more structured (e.g., ordered) data in encoded form (see [12]).

[1] *Operation-based* synchronization frameworks preserve application invariants by working in terms of a high-level, application-specific algebra of operations rather than directly manipulating replicated data. See Section 7.

We write $\mathcal{N}$ for the set of character strings and $\mathcal{T}$ for the set of unordered, edge-labeled trees whose labels are drawn from $\mathcal{N}$ and where labels of the immediate children of nodes are pairwise distinct. We draw trees sideways to save space. In text, each curly brace denotes a tree node, and each "$\mathsf{X} \mapsto$ ..." denotes a child labeled X. Also, to avoid clutter, when an edge leads to an empty tree, we usually omit the braces, the $\mapsto$ symbol, and the final childless node—e.g., "`111-1111`" actually stands for "$\{$`111-1111` $\mapsto \{\}\}$."

A tree can be viewed as a partial function from names to other trees; we write $t(n)$ for the immediate subtree of t labeled with the name n, and $dom(t)$ for the domain of a tree t—i.e. the set of the names of its immediate children. When $n \notin dom(t)$, we define $t(n)$ to be $\perp$, the "missing tree". By convention, we take $dom(\perp) = \emptyset$. To represent conflicts during synchronization, we have found it convienent to enrich the set of trees with a special "pseudo-tree" $\mathcal{X}$. We define $dom(\mathcal{X}) = \{n_{\mathcal{X}}\}$, where $n_{\mathcal{X}}$ is a special name that does not occur in ordinary trees. We write $\mathcal{T}_\perp$ for the set $\mathcal{T} \cup \{\perp\}$, $\mathcal{T}_{\mathcal{X}}$ for the set of extended trees that may contain $\mathcal{X}$ as a subtree, and $\mathcal{T}_{\mathcal{X}\perp}$ for the set $\mathcal{T}_{\mathcal{X}} \cup \{\perp\}$. Replicas are elements of $\mathcal{T}_\perp$, and archives are elements of $\mathcal{T}_{\mathcal{X}\perp}$.

A *path* is a sequence of names; the set of all paths is written $\mathcal{P}$. We write $\bullet$ for the empty path and p/q for the concatenation of paths p and q. The *projection* (or contents) of a tree, replica, or archive t at a path p, written $t(p)$, is defined as follows:

$$\begin{aligned}
t(\bullet) &= t \\
t(p) &= \mathcal{X} && \text{if } t = \mathcal{X} \\
t(n/p) &= (t(n))(p) && \text{if } t \neq \mathcal{X} \text{ and } n \in dom(t) \\
t(n/p) &= \perp && \text{if } t \neq \mathcal{X} \text{ and } n \notin dom(t)
\end{aligned}$$

Our synchronization algorithm is formulated using a semantic notion of *schemas*—a schema S is an arbitrary set of trees $S \subseteq \mathcal{T}$ (i.e., S does not include $\perp$ or $\mathcal{X}$). We write $S_\perp$ for the set $S \cup \{\perp\}$. In Section 6 we also define a syntactic notion of schema that is used for describing sets of trees in our implementation. However, the algorithm does not rely on this particular notion of schema.

3 Basic Synchronization

Harmony is designed to require only *loose coupling* with applications: it manipulates application data in external, on-disk representations such as XML trees. By contrast, many synchronizers require tight coupling between the synchronization agent and the application programs whose data is being synchronized, so that the synchronizer can see a complete trace of the operations that the application has performed on each replica, and can propagate changes by undoing and/or replaying operations. The advantage of the loosely coupled (or *state-based*) approach is that we can use Harmony to synchronize off-the-shelf applications that were implemented without replication or synchronization in mind.

Harmony's core synchronization algorithm takes two[2] current replicas and a common ancestor (all three represented as trees) and yields new replicas in which all non-conflicting changes have been merged. Suppose, for example, that we have a tree representing a small phone book:

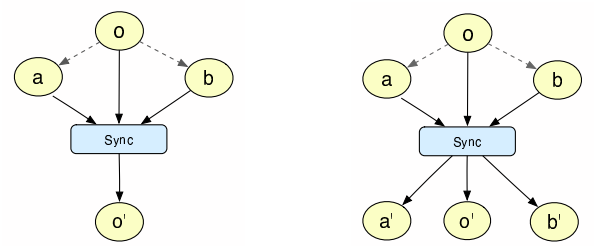
$$o = \left\{\begin{array}{l}\mathsf{Pat} \mapsto \mathtt{111\text{-}1111} \\ \mathsf{Chris} \mapsto \mathtt{222\text{-}2222}\end{array}\right\}$$

Now suppose we make two replicas of this structure, a and b, on different hosts, and separately modify one phone number in each replica:

$$a = \left\{\begin{array}{l}\mathsf{Pat} \mapsto \mathtt{111\text{-}1111} \\ \mathsf{Chris} \mapsto \mathtt{888\text{-}8888}\end{array}\right\}$$

$$b = \left\{\begin{array}{l}\mathsf{Pat} \mapsto \mathtt{999\text{-}9999} \\ \mathsf{Chris} \mapsto \mathtt{222\text{-}2222}\end{array}\right\}$$

[2] In this paper, we focus on the two-replica case. Our techniques generalize straightforwardly to synchronizing n replicas simultaneously, but the more realistic case of propagating information through a network of possibly *disconnected* replicas poses additional challenges. Our progress in this direction is described in [13].



no conflicts with conflicts

Figure 1: Synchronizer Architecture

Synchronization takes these three structures as inputs and produces a structure o' that reflects all the changes made to both replicas:

$$o' = \left\{\begin{matrix}\texttt{Pat} \mapsto \texttt{999-9999} \\ \texttt{Chris} \mapsto \texttt{888-8888}\end{matrix}\right\}$$

The original state o is provided as an input to the synchronizer so that it can tell which parts of the replicas have been updated. In the simple two-replica case that we are considering in this paper, we simply save a copy of the final merged state o' at the end of each synchronization, to use as the o the next time the synchronizer is run.[3] Another point to notice is that only the *states* of the replicas at the time of synchronization (plus the remembered state o) are available to the synchronizer: we are assuming, for the sake of loose coupling, that it has no access to the actual sequence of operations that produced a and b from o. Schematically the synchronizer may be visualized like the diagram on the left of Figure 1.

It is possible that some of the changes made to the two replicas are in conflict and cannot be merged. For example, suppose that, beginning from the same original o, we change both Pat's and Chris's phone numbers in a and, in b, delete the record for Chris entirely.

$$a = \left\{\begin{matrix}\texttt{Pat} \mapsto \texttt{123-4567} \\ \texttt{Chris} \mapsto \texttt{888-8888}\end{matrix}\right\}$$

$$b = \left\{\texttt{Pat} \mapsto \texttt{111-1111}\right\}$$

Clearly, there is no single phone book o' that incorporates both of the changes to Chris. At this point, we must choose between two evils.

1. On one hand, we can weaken users' expectations for the *persistence* of their changes to the replicas—i.e., we can decline to promise that synchronization will never lose or back out any changes that have explicitly been made to either replica. For example, here, we might choose to back out the deletion of Chris:

   $$o' = \left\{\begin{matrix}\texttt{Pat} \mapsto \texttt{999-9999} \\ \texttt{Chris} \mapsto \texttt{888-8888}\end{matrix}\right\}$$

   The user would then be notified of the lost changes and given the opportunity to re-apply them if desired.

2. Alternatively, we can keep persistence and instead give up *convergence*—i.e., we can allow the replicas to remain different after synchronization, propagating just the non-conflicting change to Pat's phone number and leaving the conflicting information about Chris untouched in each replica:

   $$a' = \left\{\begin{matrix}\texttt{Pat} \mapsto \texttt{123-4567} \\ \texttt{Chris} \mapsto \texttt{888-8888}\end{matrix}\right\}$$

   $$b' = \left\{\texttt{Pat} \mapsto \texttt{123-4567}\right\}$$

[3] In a multi-replica system, an appropriate "last shared state" would instead be calculated from the causal history of the system.

Again, the user is now notified of the conflict and manually brings the replicas back into agreement by editing one or both.

There are arguments for both alternatives. For Harmony, we have chosen the latter—favoring persistence over convergence—for two reasons. First, it is easier to specify and reason about, since it avoids making any choices about which conflicting information to retain and which to back out: it simply leaves those parts of the replicas unchanged where conflicts are discovered. Second, it gives users the possibility of temporarily ignoring conflicts and continuing to work, locally, with their replicas. By contrast, if a synchronizer backs out a change that a user has made locally, then the user *must* stop immediately and deal with the situation, or chaos can result. Section 7 discusses these trade-offs further. With this refinement, the schematic view of the synchronizer looks like the diagram on the right side of Figure 1. (The new archive o', is also an output of the synchronization process; see Section 5.)

Another important point to note about this design is that it is making decisions about *alignment*—which parts of one replica correspond to which parts of the other—completely locally, comparing edge names only at a single node from each replica. The reason, as we explain in more detail in Section 5, is that we want Harmony to be usable both interactively—reporting both proposed merges and conflicts to a human operator, who verifies (and perhaps overrides) the former and manually repairs the latter—and unsupervised, simply doing as much work as it can and leaving conflicts for later. To make this unsupervised mode safe, we need the synchronizer's behavior to be extremely conservative and easy to predict. (The cost of operating *completely* locally is that Harmony's ability to deal with list-structured data is quite limited, as we discuss in Section 6. An interesting avenue for future work is hybridizing local and non-local alignment techniques to combine their advantages; see Section 8.)

4 Conflicts

We saw in the previous section that the handling of conflicts plays a critical role in the design of a synchronizer. Before coming to the formal definition of our synchronization algorithm, we need to discuss conflicts in more depth. They come in several different forms, each affecting the behavior of the algorithm at a particular point.

Delete/Create Conflicts

The simplest form of conflict is a situation where a tree node has been deleted in one replica, while, in the other replica, a new child has been added either to it or to one of its descendants. In such cases, there is clearly no way of merging the changes into a single tree reflecting both. However, there *is* a nontrivial question of how close we want to come. For example, if the original tree and the current replicas are

$$o = \left\{ \texttt{Pat} \mapsto \left\{ \begin{array}{l} \texttt{Phone} \mapsto \texttt{333-4444} \\ \texttt{URL} \mapsto \texttt{here@there.net} \end{array} \right\} \right\}$$

$$a = \{\}$$

$$b = \left\{ \texttt{Pat} \mapsto \left\{ \begin{array}{l} \texttt{Phone} \mapsto \texttt{222-0000} \\ \texttt{URL} \mapsto \texttt{here@there.net} \end{array} \right\} \right\}$$

then it might be argued that, since nothing was changed in the subtree labeled URL in replica b and since, in replica a, this subtree got deleted, the synchronizer should propagate the deletion from a to b, leaving b' = {Pat ↦ {Phone ↦ 222-0000}}. While this behavior might be justifiable purely from the point of view of persistence of changes, we feel that users would be unhappy if synchronization could result in "partly deleted" structures like b'. Following Balasubramaniam and Pierce [3], we prefer to regard this case as a conflict at the root; our synchronization algorithm will return the original replicas unchanged.

Delete/Delete Conflicts

Another form of conflict occurs when some subtree has been deleted in one replica and one of *its* subtrees has been deleted in the other. For example:

$$o = \left\{ \texttt{Pat} \mapsto \left\{ \begin{matrix} \texttt{Phone} \mapsto \texttt{333-4444} \\ \texttt{URL} \mapsto \texttt{here@there.net} \end{matrix} \right\} \right\}$$

$$a = \{\}$$

$$b = \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \texttt{333-4444}\}\}$$

The choice to regard this situation as a conflict is not forced—one could argue that, since the changes at a are a superset of the changes at b, we should just propagate the larger deletion. However, doing so would lead to a somewhat more complex specification of the algorithm in the next section, so we have chosen here to treat this case as a conflict.

Create/Create Conflicts

The case in which different structures have been created at the same point in the two replicas is also interesting. For example:

$$o = \{\}$$

$$a = \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \texttt{333-4444}\}\}$$

$$b = \{\texttt{Pat} \mapsto \{\texttt{URL} \mapsto \texttt{here@gone.com}\}\}$$

Should this be considered a conflict, or should we merge the new substructures?

$$a' = b' = \left\{ \texttt{Pat} \mapsto \left\{ \begin{matrix} \texttt{Phone} \mapsto \texttt{333-4444} \\ \texttt{URL} \mapsto \texttt{here@gone.com} \end{matrix} \right\} \right\}$$

In contrast to the delete/delete case, it is slightly *easier*, formally, to treat such situations as non-conflicting (treating them as conflicting requires one additional clause in Definition 5.5). In practice, the situation is unclear: in the applications we have experimented with, we have found many examples where it would be inconvenient to have a conflict *and* many situations where it would be dangerous not to! Fortunately, create/create conflicts can also be handled by the mechanism of *schema conflicts*, which we introduce next. We use schemas, described below, to explicitly partition the set of create/create situations into those we should treat as conflicts and those we should not.

Schema Conflicts

The data structure on which Harmony primitively operates—unordered, edge-labeled trees—lends itself to a very straightforward recursive-tree-walking synchronization algorithm. For each node, we look at the set of child labels on each side; the ones that exist only on one side have been created or deleted (depending on the original replica), and are treated appropriately, taking into account delete/modify conflicts; for the ones that exist on both sides, we synchronize recursively. However, this procedure is too permissive: in some situations, it gives us too *few* conflicts! Consider the following example. (We revert to the fully explicit notation for trees here, to remind the reader that each "leaf value" is really just a label leading to an empty subtree.)

$$\begin{array}{rcl} o & = & \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{333-4444} \mapsto \{\}\}\}\} \\ a & = & \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{111-2222} \mapsto \{\}\}\}\} \\ b & = & \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{987-6543} \mapsto \{\}\}\}\} \end{array}$$

If we apply the naive synchronization algorithm sketched above to these replicas, we get:

$$a' = b' = \left\{ \texttt{Pat} \mapsto \left\{ \texttt{Phone} \mapsto \left\{ \begin{matrix} \texttt{111-2222} \mapsto \{\} \\ \texttt{987-6543} \mapsto \{\} \end{matrix} \right\} \right\} \right\}$$

The subtree labeled `333-4444` has been deleted in both replicas, and remains so in both `a'` and `b'`. The subtree labeled `111-2222` has been created in `a`, so we can propagate the creation to `b'` (there is no question of a create/create conflict here: this edge was created just in `a`); similarly, we can propagate the creation of `987-6543` to `a'`. But this is wrong: as far as the user is concerned, Pat's phone number was *changed* in different ways in the two replicas: what's wanted is a conflict. Indeed, if the phonebook schema only allows a single number per person, then the new replica is not only not what is wanted—it is not even well formed!

Fortunately, this last observation also contains the germ of a solution. If the synchronizer *knows* the intended schema of the structures it is synchronizing, then it can simply signal a conflict (leaving its inputs unchanged) whenever it sees that merging the changes at a particular point will lead to an ill-formed structure. We will show in Sections 6 that natural schemas can indeed be used to prevent undesired merging of changes for a wide variety of specific forms of data. But first we must formalize the synchronization algorithm itself and show in detail how it uses and manipulates schema information.

5 Synchronization

We now come to the synchronization algorithm itself. In this section, we discuss some key aspects of its design, define the algorithm formally, and relate it to a simple formal specification.

Core Requirements

We impose two fundamental requirements on synchronization: safety and maximality. Precise definitions are given below; here we just state them informally.

Safety encompasses four properties that one would expect of any reasonable synchronizer. First, it must not "back out" changes made at a replica since the last synchronization. Second, it must only propagate data between replicas and never "make up" content. Third, it must halt at conflicts and leave both replicas in their original state. Fourth, it must produce results that belong to the same schema as the originals.

However, safety alone is too weak: the trivial algorithm that always returns both replicas unchanged is safe, but doesn't synchronize anything! We say that a run of a synchronizer is *maximal* just in case it propagates as many changes as it safely can. More formally, a run is maximal iff for any path p, the results are equal at p if *any* safe run makes them equal. Thus, a maximal run incorporates all of the changes induced by any safe run. Our specification, then, for the synchronization algorithm is that every possible run must be both safe and maximal.

Locality

A fundamental consideration in the design of any synchronizer is *alignment*—i.e., identifying the parts of each replica that represent "the same information" and should be synchronized with each other.

Synchronization algorithms can be broadly grouped into two categories, according to whether they make alignment decisions *locally* or *globally*. In Section 7, we discuss a number of requirements for the generic synchronization algorithm, including simplicity, predictability, and stability. Each of these requirements strongly militates for a local approach to alignment. To see why, consider the (global) alignment decisions made by diff-based synchronization algorithms, such as the popular Unix tool `diff3`, Lindholm's 3DM [21], the work of Chawathe et al [5], and FCDP [19]. These algorithms use heuristics to make a "best guess" about what operations the user performed on the replicas by comparing the entire current states with the last common state. This works extremely well in most cases (where the best guess is clear), but in boundary cases these algorithms can make surprising decisions. This approach to alignment is fundamentally unpredictable; when alignment is determined by a complicated, global algorithm, it is difficult for a user to understand and predict how data will be aligned, especially in cases where both replicas have changed significantly. Global algorithms are also almost always unstable. For example, in the diff family of algorithms, because alignment decisions are computed from numeric metrics, very small changes to a small part of each replica can significantly impact edit distances, resulting in dramatically different alignment behavior.

To avoid these issues, our algorithm uses a simple, local alignment strategy that associates the subtrees under children with the same name with each other. The behavior of this algorithm should be easy for users to understand and predict.

Locality and Schemas

Having motivated our choice of local alignment, we turn to our requirement that the synchronizer preserve structural invariants. Because our algorithm is local, we will need a corresponding restriction to schemas that express only local constraints on structure.

As an example of a schema that expresses a *non*-local invariant, consider the following set of trees:

$$\left\{ \begin{array}{l} \{\}, \\ \{\mathsf{n} \mapsto \mathsf{x},\ \mathsf{m} \mapsto \mathsf{x}\}, \\ \{\mathsf{n} \mapsto \mathsf{y},\ \mathsf{m} \mapsto \mathsf{y},\}, \\ \{\mathsf{n} \mapsto \{\mathsf{x},\ \mathsf{y}\},\ \mathsf{m} \mapsto \mathsf{y}\}, \\ \{\mathsf{n} \mapsto \mathsf{x},\ \mathsf{m} \mapsto \{\mathsf{x},\ \mathsf{y}\}\} \end{array} \right\}$$

To see what goes wrong if we try to synchronize with respect to such a schema, consider synchronizing two trees from this set with respect to an empty archive:

$$\begin{array}{l} o = \{\} \\ a = \{\mathsf{n} \mapsto \mathsf{x},\ \mathsf{m} \mapsto \mathsf{x}\} \\ b = \{\mathsf{n} \mapsto \mathsf{y},\ \mathsf{m} \mapsto \mathsf{y}\} \end{array}$$

A local synchronization algorithm that aligns by name will recursively synchronize the subtrees under names n and m. However, it is not obvious which *schema* we should use for each of these recursive calls, because the set of trees that can validly appear under n depends on the subtree under m and vice versa. We might try the schema that consists of all the trees that can appear under n (and similarly for m): $\{\mathsf{x},\ \mathsf{y}, \{\mathsf{x},\ \mathsf{y}\}\}$. With this schema, the synchronizer computes the tree $\{\mathsf{x},\ \mathsf{y}\}$ for both n and m, reflecting the fact that x and y were both added under n and m. However, the trees cannot be assembled into a single well-formed tree, because the result,

$$\left\{ \begin{array}{l} \mathsf{n} \mapsto \{\mathsf{x},\ \mathsf{y}\}, \\ \mathsf{m} \mapsto \{\mathsf{x},\ \mathsf{y}\} \end{array} \right\}$$

is not in the original schema. The "most synchronized" well-typed results are actually

$$a' = \left\{ \begin{array}{l} \mathsf{n} \mapsto \mathsf{x}, \\ \mathsf{m} \mapsto \{\mathsf{x},\ \mathsf{y}\} \end{array} \right\} \quad b' = \left\{ \begin{array}{l} \mathsf{n} \mapsto \{\mathsf{x},\ \mathsf{y}\}, \\ \mathsf{m} \mapsto \mathsf{y} \end{array} \right\}$$

but there does not seem to be any way to find them without backtracking.

The main problem with this schema is that it expresses a global invariant—at most one of n or m may have $\{\mathsf{x},\ \mathsf{y}\}$ as subtree—which cannot be easily preserved by a local synchronization algorithm. To avoid these situations, we impose a restriction on schemas, which we call *path consistency*, that is analogous to the restriction on full-blown tree grammars embodied by W3C Schema. Intuitively, a schema is path consistent if any subtree that can validly appear at some path in one tree can validly be "transplanted" to the same location in any other tree with the same schema. This restriction ensures that the schema that we use to recursively synchronize a single child is consistent across the entire schema; i.e., the set of trees that may validly appear under a child does not depend on the presence (or absence) of other parts of the tree.

To define path consistency precisely, we need a little new notation. First, the notion of projection at a path is extended pointwise to schemas—that is, for a schema $S \subseteq \mathcal{T}$ and path $p \in \mathcal{P}$, we have $S(p) = \{t(p) \mid t \in S \wedge t(p) \neq \bot\}$. Note that the projection of a schema at any path is itself a schema. Next, we define what it means to transplant a subtree from one tree to another at a given path.

5.1 Definition [Path Update]: Let t be a tree and p a path such that $t(p) \in \mathcal{T}$. We define the update of t at p with t', written $t[p \mapsto t']$, inductively on the structure of p as:

$$t[\bullet \mapsto t'] = t'$$
$$t[n/p \mapsto t'] = \left\{ \begin{array}{ll} n \mapsto t(n)[p \mapsto t'] & \\ m \mapsto t(m) & \text{for } m \in dom(t) \setminus \{n\} \end{array} \right\}$$

Now, a schema S is path consistent if, whenever t and t' are in S, then, for every path p, the result of updating t along p with $t'(p)$ is also in the schema.

5.2 Definition [Path Consistency]: A schema S is path consistent iff, for all $t, t' \in S$ and $p \in \mathcal{P}$, we have

$$t(p) \neq \bot \ \wedge \ t'(p) \neq \bot \implies t[p \mapsto t'(p)] \in S.$$

Maximality and Schema Conflicts

There is one final complication that arises in schema-aware synchronization algorithms: on some inputs, there *are no* safe and maximal runs belonging to the schema. Consider a run of a synchronizer on the following three trees:

$$\begin{array}{rcl} o & = & \{\mathsf{v}\} \\ a & = & \{\mathsf{w}, \mathsf{y}, \mathsf{z}\} \\ b & = & \{\mathsf{w}, \mathsf{x}\} \end{array}$$

with respect to the following schema:

$$\left\{ \begin{array}{c} \{\mathsf{v}\}, \{\mathsf{w}, \mathsf{x}\}, \{\mathsf{w}, \mathsf{x}, \mathsf{y}\}, \\ \{\mathsf{w}, \mathsf{x}, \mathsf{z}\}, \{\mathsf{w}, \mathsf{y}, \mathsf{z}\} \end{array} \right\}$$

For the a replica, the only safe result that belongs to the schema is $a' = a$. However, on the b side, there are three possible safe results belonging to the schema, and none are maximal:

$$\begin{array}{c} \{\mathsf{w}, \mathsf{x}\} \\ \{\mathsf{w}, \mathsf{x}, \mathsf{y}\} \\ \{\mathsf{w}, \mathsf{x}, \mathsf{z}\} \end{array}$$

Notice that, since the tree $\{\mathsf{w}, \mathsf{x}, \mathsf{y}, \mathsf{z}\}$ does not belong to the schema, we cannot include both x and y in b' (without backing out the addition of z). Indeed, for each choice of b', there is a path p where b' at p is different from a' at p, but for a different choice of b', the trees at those paths are equal. Hence, none of the safe runs belonging to the schema are maximal.

To ensure that synchronization always has a maximal, safe result, we strengthen the notion of safety by introducing a new sort of conflict. Informally, a *schema domain conflict* is produced whenever propagating *all* of the (otherwise non-conflicting) additions and deletions of children at a node yields an ill-formed result (see Definition 5.6 below for the full formal definition). For example, on the above trees, our algorithm yields a schema domain conflict at the root since it cannot add both x and y to a'.

As a side-note, we should remark that the approach presented in this section is not the only way to ensure maximality in a schema-directed algorithm; we have considered several alternatives. First, we could throw out schema conflicts and require instead that schemas be closed under the "shuffling" of their domains with the domains of other trees in the schema. This approach amounts to declaring, by fiat, that the maximal result of every possible synchronization must be present in the schema. For example, in the schema above, we would need to include the tree $\{\mathsf{w}, \mathsf{x}, \mathsf{y}, \mathsf{z}\}$ in the schema. We have not pursued this idea because it does not appear that these shuffled schemas would be flexible enough to express the kinds of invariants needed in the applications we are considering. Alternatively, we could recognize schema domain conflicts as here, but, instead of requiring that the replicas remain unchanged, only require that the *domains* be unchanged. This approach would allow "deep" synchronization of subtrees, which has some advantages, but we have found it hard to reason about the results of synchronization. For these reasons, our first—simplest—proposal seems best.

$sync(S, o, a, b) =$
  if $a = b$ then (a, a, b) – *equal replicas: done*
  else if $a = o$ then (b, b, b) – *no change to a: propagate b*
  else if $b = o$ then (a, a, a) – *no change to b: propagate a*
  else if $o = \mathcal{X}$ then (o, a, b) – *unresolved conflict*
  else if $a = \perp$ then $(\mathcal{X}, a, b)$ – *delete/modify conflict*
  else if $b = \perp$ then $(\mathcal{X}, a, b)$ – *delete/modify conflict*
  else – *proceed recursively*
    let $(o'(k), a'(k), b'(k)) = sync(S(k), o(k), a(k), b(k))$
      $\forall k \in dom(a) \cup dom(b)$
    in if $(dom(a') \notin doms(S))$ or $(dom(b') \notin doms(S))$
      then $(\mathcal{X}, a, b)$ – *schema conflict*
      else (o', a', b')

Figure 2: Core Synchronization Algorithm

Synchronization Algorithm

The core synchronization algorithm is depicted in Figure 2. In outline, its structure is as follows: we first check for trivial cases (replicas being equal to each other or unmodified), then we check for delete/modify conflicts, and in the general case we recurse on each child label and check for schema conflicts before returning the results. In practice, synchronization will be performed repeatedly, with additional updates applied to one or both of the replicas between synchronizations. To support this, a new archive needs to be constructed by the synchronizer. Its calculation is straightforward: we use the synchronized version at every path where the replicas agree and insert a conflict marker $\mathcal{X}$ at paths where the replicas are in conflict.

Formally, the algorithm takes as inputs a path consistent schema S, an archive o, and two current replicas a and b; it outputs a new archive o' and two new replicas a' and b'. We require that both a and b belong to $S_\perp$, (recall that $\perp$ stands for a deleted replica). Both the input and output archives may contain the special conflict tree $\mathcal{X}$. The algorithm also relies on one piece of new notation: $doms(S)$ stands for the *domain-set* of S; i.e., the set of all domains of trees in S—i.e., $doms(S) = \{dom(t) \mid t \in S\}$.

In the case where a and b are identical (they are both the same tree or both $\perp$), they are immediately returned, and the new archive is set to their value. If one of the replicas is unchanged (equal to the archive), then all the changes in the other replica can safely be propagated, so we simply return three copies of it as the result replicas and archive. Otherwise, both replicas have changed, in different ways. In this case, if the archive is a conflict, then the conflict is preserved and a and b are returned unmodified. If one replica is missing (it has been deleted), then we have a *delete/modify conflict* (either *delete/create* or *delete/delete*) since the other replica has changed, so we simply return the original replicas.

Finally, in the general case, the algorithm recurses: for each k in the domain of either current replica, we call $sync$ with the corresponding subtrees from o, a, and b (any of which may be $\perp$); we collect up the results of these calls to form new trees o', a', and b'. If either of the new replicas is ill formed (i.e., its domain is not in the domain-set of the schema), then we have a schema domain conflict and the original replicas are returned unmodified. Otherwise, the synchronized results are returned.

Specification

We now give a formal specification of the properties we want our synchronization algorithm to satisfy. Our presentation follows the basic approach used for specifying the Unison file synchronizer [29].

We start by with a few auxiliary definitions which are needed in the formal statements of safety and maximality. Both safety and maximality are based on a notion of *local equivalence*, which relates two trees (or replicas or archives) if their top-level nodes are similar—i.e., intuitively, if both are present, both are missing, or both are conflicting.

5.3 Definition [Local equivalence]: We say that two elements of $\mathcal{T}_{\mathcal{X}\perp}$ are locally equivalent, written $t \sim t'$, iff

- $t = t' = \mathcal{X}$ or
- $t = t' = \perp$ or
- $t \in \mathcal{T}$ and $t' \in \mathcal{T}$.

5.4 Lemma: The local equivalence relation is an equivalence.

Proof: The definition is obviously reflexive and symmetric. For transitivity, choose any $t, t', t'' \in \mathcal{T}_{\mathcal{X}\perp}$ such that $t \sim t'$ and $t' \sim t''$. We show $t \sim t''$ by cases on the local equivalence rule applied to derive $t \sim t'$.

- If $t = t' = \mathcal{X}$, then as $t' \sim t''$ we must have $t'' = \mathcal{X}$, hence $t \sim t''$.
- If $t = t' = \perp$, then as $t' \sim t''$ we must have $t'' = \perp$, hence $t \sim t''$.
- If both $t \in \mathcal{T}$ and $t' \in \mathcal{T}$, then by $t' \sim t''$, we know that t'' is not $\perp$ and not $\mathcal{X}$, hence $t \sim t''$. □

In the following we silently rely on the fact that $\sim$ is an equivalence relation.

The definition of safety relies on the notion of conflict. We use local equivalence to capture all the simple notions of conflicts that consider only the presence or absence of a single node.

5.5 Definition [Local Conflict]: We say that an archive o and two replicas a, b are locally conflicting, written $\mathsf{localconflict}(o, a, b)$, if

- $(o = \mathcal{X}) \wedge (a \neq b)$ or
- $(o \neq a) \wedge (o \neq b) \wedge (a \nsim b)$

Intuitively, replicas a and b are locally conflicting if there is a conflict recorded in the archive o that has not been resolved, or if they have both changed since the state recorded in the archive but are not locally equivalent. The conflicts described in Section 4, with the exception of schema conflicts, are captured by the definitions of local equivalence and local conflict.

Next we define schema domain conflicts formally. As described above, schema domain conflicts can be detected using a local test on the domain of the tree being synchronized.

5.6 Definition [Schema Domain Conflict]: Let $S \subseteq \mathcal{T}$ be a schema. We say that an archive o, and two replicas a, b have a schema domain conflict iff

$$(a \neq \perp) \ \wedge \ (b \neq \perp) \wedge (newdom_a \notin doms(S) \vee newdom_b \notin doms(S))$$

where the new domains are defined as

$$\begin{array}{rcl} newdom_a & = & \{k \in dom(a) \mid o(k) \neq a(k)\} \\ & \cup & \{k \in dom(b) \mid o(k) = \perp\} \\ & \cup & (dom(a) \cap dom(b)) \\ \\ newdom_b & = & \{k \in dom(b) \mid o(k) \neq b(k)\} \\ & \cup & \{k \in dom(a) \mid o(k) = \perp\} \\ & \cup & (dom(a) \cap dom(b)). \end{array}$$

Intuitively, a schema domain conflict occurs when one of the new domains $newdom_a$ and $newdom_b$ is not in the domain-set of S. The new domain $newdom_a$ is the expected domain of the tree that results from safely propagating into replica a the changes from replica b. It contains all names under which there is a change in replica a, all names added by replica b, and all the names preserved in both a and b. The new domain $newdom_b$ is defined similarly, reversing the roles of a and b.

Given definitions of local conflict and schema domain conflict, we can now define conflicts. Given a schema S, we say that an archive o, and two replicas a, b are conflicting, written $\mathsf{conflict}(S, o, a, b)$, iff they are either locally conflicting or schema domain conflicting:

5.7 Definition [Conflict]:

$$\mathsf{conflict}(S, o, a, b) = \mathsf{localconflict}(o, a, b) \vee \mathsf{schemadomconflict}(S, o, a, b).$$

Safety and maximality are both properties of *runs*.

5.8 Definition [Run]: A *run* of a synchronizer is a tuple (S, o, a, b, o', a', b') of a schema S and six trees, representing the original synchronized state (o), the states of the two replicas before synchronization (a, b), the new archive (o'), and the states of the replicas after synchronization (a', b').

A *safe* run of a synchronizer satisfies the following safety properties: the result of synchronization must reflect all user changes, it must not include changes that do not come from either replica, trees under a conflicting node should remain untouched, and the results should belong to the schema.

5.9 Definition [Locally Safe Run]: A run is said to be *locally safe* iff:

1. It never overwrites changes locally:

$$(o \nsim a \implies a' \sim a) \ \wedge \ (o \nsim b \implies b' \sim b)$$

2. It never "makes up" content locally:

$$\begin{array}{rcl} a \nsim a' & \implies & b \sim a' \\ b \nsim b' & \implies & a \sim b' \\ o' \neq \mathcal{X} & \implies & o' \sim a' \wedge o' \sim b' \end{array}$$

3. It stops at conflicting paths (leaving replicas in their current states and recording the conflict):

$$\mathsf{conflict}(o, a, b) \implies (a' = a) \wedge (b' = b) \wedge (o' = \mathcal{X})$$

4. It yields results belonging to the schema (or missing):

$$(a' \in S_\perp) \ \wedge \ (b' \in S_\perp)$$

5.10 Definition [Safe Run]: A run (S,o,a,b,o',a',b') is *safe*, written $safe(S,o,a,b,o',a',b')$, iff for every path p, the sub-run $(S(p),o(p),a(p),b(p),o'(p),a'(p),b'(p))$ is locally safe.

5.11 Definition [Maximal Run]: A run $(S,o,a,b,\ o',a',b')$ is *maximal* iff it is safe and propagates at least the same changes as any other safe run, i.e.

$$\forall o'', a'', b''.\ safe(S, o, a, b, o'', a'', b'') \implies \begin{cases} \forall p \in \mathcal{P}.\ a''(p) \sim b''(p) \implies a'(p) \sim b'(p) \\ \forall p \in \mathcal{P}.\ o''(p) \neq \mathcal{X} \quad \implies o'(p) \neq \mathcal{X}. \end{cases}$$

We can now state precisely what we mean by claiming that Harmony's synchronization algorithm is correct.

5.12 Theorem: Let $S \subseteq \mathcal{T}$ be a path-consistent schema. If $a, b \in S_\perp$ and $\mathtt{sync}(S, o, a, b)$ evaluates to (o', a', b'), then (S, o, a, b, o', a', b') is maximal.

In the proofs we often proceed by induction on the height of a tree. We define $height(\perp) = height(\mathcal{X}) = 0$ and the height of any other tree to be $height(t) = 1 + \mathtt{max}(\{height(t(k)) \mid k \in dom(t)\})$. Note that the height of the empty tree (a node with no children) is 1, to avoid confusing it with the missing or the conflict tree.

Before proving Theorem 5.12, we prove a few technical lemmas. First we note that path consistency is stable under projection on a single name.

5.13 Lemma: Let S be a path consistent schema. For any name n, $S(n)$ is path consistent.

Proof: Let t_n and t'_n be trees in $S(n)$, and p a path with $t_n(p) \in \mathcal{T}$ and $t'_n(p) \in \mathcal{T}$. As t_n and t'_n are in $S(n)$, there exist t and t' in S with $t(n) = t_n$ and $t'(n) = t'_n$. As S is path consistent and $t(n/p) = t_n(p) \in \mathcal{T}$ and $t'(n/p) = t'_n(p) \in \mathcal{T}$, we have

$$\begin{aligned} t[n/p \mapsto t'(n/p)] &= t[n/p \mapsto t'_n(p)] \\ &= \left\{ \begin{array}{l} n \mapsto t(n)[p \mapsto t'_n(p)] \\ m \mapsto t(m) \quad \text{for } m \in dom(t) \setminus \{n\} \end{array} \right\} \\ &= \left\{ \begin{array}{l} n \mapsto t_n[p \mapsto t'_n(p)] \\ m \mapsto t(m) \quad \text{for } m \in dom(t) \setminus \{n\} \end{array} \right\} \\ &\in S, \end{aligned}$$

hence $t_n[p \mapsto t'_n(p)] \in S(n)$. □

The next lemma illustrates the key property enjoyed by path consistent schemas: that we can assemble well-formed trees (belonging to the schema) out of well-formed subtrees. This property tells us that the trees produced in the recursive case of the synchronization algorithm are correct.

5.14 Lemma: Let S be a path consistent schema. If $dom(t) \in doms(S)$ and for each $k \in dom(t)$ we have $t(k) \in S(k)$ then $t \in S$.

Proof: Let $dom(t) = \{n_1, \ldots, n_j\}$. As $t(k) \in S(k)$ for every $k \in dom(t)$, for each $k \in \{n_1 \ldots n_j\}$ there exists a tree $t'_k \in S$ with $t'_k(k) = t(k)$. Moreover, as $dom(t) \in doms(S)$, there is a tree $t'' \in S$ with $dom(t'') = dom(t)$.

We now show by a finite induction on $i \leq j$ that $t''[n_1 \mapsto t'_{n_1}(n_1)] \ldots [n_i \mapsto t'_i(n_i)] \in S$.

The base case, $i = 1$, is immediate by path consistency, as both t'' and t'_1 are in S, and as both $t''(n_1) \in \mathcal{T}$ and $t'_1(n_1) \in \mathcal{T}$.

Consider now $i > 1$. By the inner induction hypothesis, we have

$$t''' = t''[n_1 \mapsto t'_1(n_1)] \ldots [n_{i-1} \mapsto t'_{i-1}(n_{i-1})] \in S$$

By hypothesis, we have $t'_i \in S$. Moreover, both $t'''(n_i) = t''(n_i) \in \mathcal{T}$ and $t'_k(n_i) \in \mathcal{T}$. By path consistency for S, we hence have $t'''[n_i \mapsto t'_i(n_i)] \in S$.

We conclude that $t = t''[n_1 \mapsto t'_1(n_1)] \ldots [n_j \mapsto t'_j(n_j)] \in S$. □

Next we prove a lemma that records some examples of safe runs (both to illuminate the definitions and to avoid some redundancy in the proof of Theorem 5.12.).

5.15 Lemma: If $a \in S_\perp$ and $b \in S_\perp$ then the following runs

1. $(S, o, a, b, \mathcal{X}, a, b)$
2. (S, o, a, a, a, a, a)
3. (S, o, o, b, b, b, b)
4. (S, o, a, o, a, a, a)

are safe.

Proof: Let p be a path.

(1) We must check that the run $(S, o, a, b, \mathcal{X}, a, b)$ is locally safe at p. We have $\mathcal{X}(p) = \mathcal{X}$. As $a'(p) = a(p)$, $b'(p) = b(p)$, and $o'(p) = \mathcal{X}$, local safety conditions (1,2) are satisfied. Local safety condition (3) is trivially satisfied as $a'(p) = a(p)$, $b'(p) = b(p)$, and $o'(p) = \mathcal{X}$. The final condition (4) is satisfied since, using the definition of schema projection, we have $a'(p) = a(p) \in S(p)_\perp$ and $b'(p) = b(p) \in S(p)_\perp$.

(2) We must check that the run (S, o, a, a, a, a, a) is locally safe at p. Local safety condition (1) is satisfied since $a'(p) = a(p) \sim a(p)$ and $b'(p) = a(p) = b(p) \sim b(p)$. Local safety condition (2) is satisfied because $a(p) \sim a(p) = a'(p)$ and $o'(p) = a(p) \sim a(p) = a'(p)$ and similarly for $b(p)$. Local safety condition (3) is satisfied because there cannot be a conflict at path p. The first condition of local conflict is not satisfied because $a(p) = b(p)$ and the second condition of local conflict is not satisfied because $a(p) = b(p) \sim b(p)$, so there is no local conflict at path p. We address schema domain conflict by computing as follows:

$$\begin{array}{lcll} newdom_{a(p)} & = & \{k \in dom(a(p)) \mid o(p/k) \neq a(p/k)\} & \left(\subseteq dom(a(p)) \right) \\ & \cup & \{k \in dom(a(p)) \mid o(p/k) = \bot\} & \left(\subseteq dom(a(p)) \right) \\ & \cup & (dom(a(p)) \cap dom(a(p))) & \left(= dom(a(p)) \right) \\ & = & dom(a(p)) & \end{array}$$

$$\begin{array}{lcll} newdom_{b(p)} & = & \{k \in dom(a(p)) \mid o(p/k) \neq a(p/k)\} & \left(\subseteq dom(a(p)) \right) \\ & \cup & \{k \in dom(a(p)) \mid o(p/k) = \bot\} & \left(\subseteq dom(a(p)) \right) \\ & \cup & (dom(a(p)) \cap dom(a(p))) & \left(= dom(a(p)) \right) \\ & = & dom(a(p)) & \end{array}$$

If $a(p) = \bot$ then by definition, there is no schema domain conflict at path p. Otherwise, $a(p) \neq \bot$ and with $a \in S_\bot$ we have that $newdom_{a(p)} = newdom_{b(p)} = dom(a(p)) \in doms(S(p))$. Hence, there is no schema domain conflict at path p. The final local safety condition (4) is satisfied as $a'(p) = b'(p) = a(p) \in S(p)_\bot$ by the definition of schema projection.

(3) We must check that (S, o, o, b, b, b, b) is locally safe at p. Local safety condition (1) is satisfied because $o(p) = a(p) \sim a(p)$ and $b(p) = b'(p) \sim b'(p)$. Local safety condition (2) is satisfied because $o'(p) = b(p) \sim b(p) = a'(p) = b'(p)$. Local safety condition (3) is satisfied because there cannot be a conflict at path p. The first condition of local conflict is not satisfied because $o(p) = a(p) \neq \mathcal{X}$ and the second condition of local conflict is not satisfied because $o(p) = a(p)$, so there is no local conflict at path p. As before, we address schema domain conflict by computing as follows:

$$\begin{array}{lcll} newdom_{a(p)} & = & \{k \in dom(o(p)) \mid o(p/k) \neq o(p/k)\} & \left(= \emptyset \right) \\ & \cup & \{k \in dom(b(p)) \mid o(p/k) = \bot\} & \left(\supseteq dom(b(p)) \setminus dom(o(p)) \right) \\ & \cup & (dom(o(p)) \cap dom(b(p))) & \\ & = & dom(b(p)) & \end{array}$$

$$\begin{array}{lcll} newdom_{b(p)} & = & \{k \in dom(b(p)) \mid o(p/k) \neq b(p/k)\} & \left(\supseteq dom(b(p)) \setminus dom(o(p)) \right) \\ & \cup & \{k \in dom(o(p)) \mid o(p/k) = \bot\} & \left(= \emptyset \right) \\ & \cup & (dom(o(p)) \cap dom(b(p))) & \\ & = & dom(b(p)) & \end{array}$$

If $b(p) = \bot$ then by definition, there is not a schema domain conflict at path p. Otherwise, $b(p) \neq \bot$ and as $b \in S_\bot$ we have that $newdom_{b(p)} = newdom_{a(p)} = dom(b(p)) \in doms(S(p))$. Hence, there is no schema domain conflict at path p. The final local safety condition (4) is satisfied as $a'(p) = b'(p) = b(p) \in S(p)_\bot$ using the definition of schema projection.

(4) Symmetric to the previous case, inverting the roles of a and b. □

Now we prove the main theorem.

Proof of Theorem 5.12: The proof goes by induction on the sum of the depth of o, a, and b, with a case analysis according to the first rule in the algorithm that applies.

Case $a = b$**:** $o' = a \quad a' = a \quad b' = b$

By Lemma 5.15(2) the run (S, o, a, a, a, a, a) is safe.

We must now show that (S, o, a, a, a, a, a) is maximal. The first condition for maximality is immediate as for all paths p, we have $a'(p) \sim b'(p)$. The second condition is also satisfied, since $o' = a$, hence we have $o'(p) \neq \mathcal{X}$ for all paths p.

Case $a = o$**:** $o' = b \quad a' = b \quad b' = b$

By Lemma 5.15(3) the run (S, o, o, b, b, b, b) is safe.

We must now show that (S, o, o, b, b, b, b) is maximal. The first condition for maximality is immediate, since $a'(p) \sim b'(p)$ for all paths p. The second condition is also satisfied, since $o' = b$, hence we have $o'(p) \neq \mathcal{X}$ for all paths p.

Case $b = o$**:** $o' = a \quad a' = a \quad b' = a$

Symmetric to the previous case, inverting the roles of a and b.

Case $o = \mathcal{X}$**:** $o' = \mathcal{X} \quad a' = a \quad b' = b$

By Lemma 5.15(1) the run $(S, \mathcal{X}, a, b, \mathcal{X}, a, b)$ is safe.

We must now show that the run is maximal. The predicate $\mathsf{localconflict}(\mathcal{X}, a, b)$ is satisfied because we know that $o = \mathcal{X}$ and $a \neq b$ (as the first case of the algorithm did not apply). Therefore, by safety condition 3, the only safe run is $(S, \mathcal{X}, a, b, \mathcal{X}, a, b)$, hence it is maximal.

Case $a = \bot$**:** $o' = \mathcal{X} \quad a' = \bot \quad b' = b$

By Lemma 5.15(1) the run $(S, o, \bot, b, \mathcal{X}, \bot, b)$ is safe.

We must now prove that the run is maximal. None of the previous rules apply, so we must have $b \neq a = \bot$, $o \neq a = \bot$, and $b \neq o$. Since $a = \bot$ and $b \neq \bot$, we have $a \nsim b$. Hence the predicate $\mathsf{localconflict}(o, a, b)$ is satisfied. As before, by safety condition 3, the only safe run is $(S, \mathcal{X}, a, b, \mathcal{X}, a, b)$, hence it is maximal.

Case $b = \bot$**:** $o' = \mathcal{X} \quad a' = a \quad b' = \bot$

Symmetric to the previous case, inverting the roles of a and b.

Recursive case: Since previous cases of the algorithm do not apply, we have $a \neq b$, $o \neq \mathcal{X}$, $a \neq \bot$, $b \neq \bot$, and $a \sim b$.

By Lemma 5.13, each of the schemas $S(k)$ are path consistent for $k \in dom(a) \cup dom(b)$. By the definition of schema projection, for each k we have $a(k), b(k) \in S(k)_\bot$. Thus, by the induction hypothesis (which we may apply as neither a nor b is missing, hence the sum of the depths decreases), each recursive sub-run is maximal. In particular, each sub-run is safe and gives results in $S(k)_\bot$.

Let o'_r, a'_r, and b'_r be trees obtained by the recursive calls, i.e. $(o'_r(k), a'_r(k), b'_r(k)) = \mathtt{sync}(o(k), a(k), b(k))$ for all $k \in dom(a) \cup dom(b)$. From the facts obtained by the IH and the definition of safety, it follows that the run $(S, o, a, b, o'_r, a'_r, b'_r)$ is locally safe at every path except possibly at the root. We now check that it is also locally safe at the empty path and maximal.

First we consider local safety. We start by showing that $dom(a'_r) = newdom_a$. For this let $k \in dom(a) \cup dom(b)$ and consider the shape of possible tuples $(o(k), a(k), b(k))$. We let the variables t_o, t_a, t_b range over elements of $\mathcal{T}$:

$(\mathcal{X}, \bot, \bot)$**:** Cannot occur because $k \in dom(a) \cup dom(b)$.

$(\bot, \bot, \bot)$**:** Same as previous case.

$(t_o, \bot, \bot)$**:** Same as previous case.

$(\bot, t_a, \bot)$**:** Since $o(k) \nsim a(k)$ we get by local safety condition (1) that $a'_r(k) \sim t_a$ and in particular $a'_r(k) \neq \bot$, hence $k \in dom(a'_r)$. On the other hand, since $k \in \{n \in dom(a) \mid o(n) \neq a(k)\}$ we also have that $k \in newdom_a$.

$(\bot, t_a, t_b)$**:** Same as previous case.

$(\mathcal{X}, t_a, \bot)$**:** Cannot occur because $o \neq \mathcal{X}$.

$(\mathcal{X}, t_a, t_b)$**:** Same as previous case.

$(t_o, \bot, t_b)$**:** Since $o(k) \nsim a(k)$ we get by local safety condition (1) that $a'_r(k) \sim \bot$ and therefore $a'_r(k) = \bot$, so $k \notin dom(a'_r)$. On the other hand, since $a(k) = \bot$ we have that $k \notin dom(a)$ so especially $k \notin \{n \in dom(a) \mid o(n) \neq a(n)\}$ and $k \notin dom(a) \cap dom(b)$, and since $o(k) \neq \bot$ we have that $k \notin \{n \in dom(b) \mid o(n) = \bot\}$, hence also $k \notin newdom_a$.

$(\mathcal{X}, \bot, t_b)$**:** Cannot occur because $o \neq \mathcal{X}$.

$(\bot, \bot, t_b)$**:** By Lemma 5.15(3) the run $(S(k), \bot, \bot, t_b, t_b, t_b, t_b)$ is safe. Letting a'' and b'' stand for the results of this run, which are both t_b, we have that $a''(k) \sim b''(k)$. By the maximality of the recursive sub-run we get that $a'_r(k) \sim b'_r(k)$. As $t_b \nsim \bot$, we have $b'_r(k) \sim t_b$ thus $b'_r(k) \neq \bot$ by local safety (1), hence $a'_r(k) \neq \bot$. From this, we have $k \in dom(a'_r)$. On the other hand, since $k \in \{n \in dom(b) \mid o(n) = \bot\}$ we also have that $k \in newdom_a$.

$(t_o, t_a, \bot)$**:** We first consider the case $t_o \neq t_a$. This means there is a path p such that $o(k/p) \nsim a(k/p)$ so we get by local safety condition (1) that $a'_r(k/p) \sim a(k/p)$ and especially $a'_r(k) \neq \bot$, so $k \in dom(a'_r)$. On the other hand, since $o(k) \neq a(k)$ we have that $k \in \{n \in dom(a) \mid o(n) \neq a(n)\}$, hence also $k \in newdom_a$.

Now, we consider the case $t_o = t_a$. By Lemma 5.15(3) the run $(S(k), t_o, t_o, \bot, \bot, \bot, \bot)$ is safe. Letting a'' and b'' stand for the results of this run, which are both $\bot$, we have that $a''(k) \sim b''(k)$. Therefore, by maximality of the recursive sub-run we get that $a'_r(k) \sim b'_r(k)$. Next we show that $a'_r(k) = \bot$. Since $t_o \nsim \bot$ we have that $b'_r(k) \sim \bot$ by local safety (1) and hence $a'_r(k) = \bot$. It follows that $k \notin dom(a'_r)$. On the other hand, since $o(k) = a(k)$ we have $k \notin \{n \in dom(a) \mid o(n) \neq a(n)\}$, and since $k \notin dom(b)$ we have $k \notin dom(a) \cap dom(b)$ and $k \notin \{n \in dom(b) \mid o(n) = \bot\}$, hence also $k \notin newdom_a$.

(t_o, t_a, t_b)**:** We first consider the case $t_o \neq t_a$. This means there is a path p such that $o(k/p) \nsim a(k/p)$ so we get by local safety condition (1) that $a'_r(k/p) \sim a(k/p)$ and especially $a'_r(k) \neq \bot$, so $k \in dom(a'_r)$. On the other hand, since $o(k) \neq a(k)$ we have that $k \in \{n \in dom(a) \mid o(n) \neq a(n)\}$, hence also $k \in newdom_a$.

Now, we consider the case $t_o = t_a$. By Lemma 5.15(3) the run $(S(k), t_o, t_o, t_b, t_b, t_b, t_b)$ is safe. Letting a'' and b'' stand for the results of this run, which are both t_b, we have that $a''(k) \sim b''(k)$. Therefore, by the maximality of the recursive sub-run (IH) we get that $a'_r(k) \sim b'_r(k)$. Next we show that $a'_r(k) \neq \bot$. We consider two subcases. If $a'_r(k) \sim t_a$ then $a'_r(k) \neq \bot$ by definition. The other case, $a'_r(k) \nsim t_a$, is a contradiction since by local safety (2), $a'_r(k) \sim t_b$, and since $t_a, t_b \in \mathcal{T}$ we also have $a'_r(k) \sim t_a$. It follows that $k \in dom(a'_r)$. On the other hand, since $k \in dom(a) \cap dom(b)$ we also have $k \in newdom_a$.

By a symmetric argument we can show that $dom(b'_r) = newdom_b$.

We first cover the case where $dom(a'_r) \notin doms(S)$ or $dom(b'_r) \notin doms(S)$. In this case $o' = \mathcal{X}$, $a' = a$, and $b' = b$ and it follows immediately from Lemma 5.15(1) that the run $(\mathcal{X}, a, b, \mathcal{X}, a, b)$ is safe. To show that it is maximal we show that there is a schema domain conflict at the root. We have that $newdom_a = dom(a'_r) \notin doms(S)$ or $newdom_b = dom(b'_r) \notin doms(S)$, so $\mathsf{schemadomconflict}(S, o, a, b)$ holds. As above, by safety condition (3), the only safe run is $(\mathcal{X}, a, b, \mathcal{X}, a, b)$, hence it is maximal.

We now cover the case where $dom(a'_r) \in doms(S)$ and $dom(b'_r) \in doms(S)$. In this case $o' = o'_r$, $a' = a'_r$, and $b' = b'_r$ and since these are built as the results of the recursive calls, we have $o' \neq \bot$, $a' \neq \bot$, and $b' \neq \bot$ (recall the difference between the empty tree and the missing tree). It follows directly that $a \sim a'$, $b \sim b'$, $a' \sim b'$, $o' \sim a'$, $b \sim b'$ and $o' \sim b'$, which immediately satisfies local safety conditions (1, 2). Local safety condition (3) is satisfied because there cannot be a conflict at the root: there is no local conflict because $o \neq \mathcal{X}$ and $a \sim b$, and there is no schema domain conflict because $newdom_a = dom(a'_r) \in doms(S)$ and $newdom_b = dom(b'_r) \in doms(S)$. For the final safety condition (4), we show that $a', b' \in S_\bot$. As each sub-run is maximal (and hence, safe), for every $k \in dom(a')$ we have $a'(k) \in S(k)_\bot$. Also, since $k \in dom(a')$ we have $a'(k) \neq \bot$ and so $a'(k) \in S(k)$. We also have that $dom(a') \in doms(S)$. By Lemma 5.14, $a' \in S$ and hence $a' \in S_\bot$. By a symmetric argument, we have $b' \in S_\bot$. We conclude that the run is locally safe at the root.

To finish the proof, we must also show that the run is maximal. Now, let $(S, o, a, b, o'', a'', b'')$ be another safe run and let p be a path.

- Let us first consider the case where p is the empty path.

- We have $a' \sim b'$, so the first maximality condition is satisfied at the root.
- We have $o' \neq \mathcal{X}$, so the second maximality condition is satisfied at the root

- If p is not the empty path, then it may be decomposed as k/p'. By IH, the run $(S(k), o(k), a(k), b(k), o'(k), a'(k), b'(k))$ is maximal, and by the definition of a safe run, the sub-run $(S(k), o(k), a(k), b(k), o''(k), a''(k), b''(k))$ is also a safe run. We have $a''(p) = a''(k/p') = (a''(k))(p')$, and $b''(p) = b''(k/p') = (b''(k))(p')$.
  - If $a''(p) \sim b''(p)$, then $a''(p) = (a''(k))(p') \sim (b''(k))(p') = b''(p)$, hence we have (by maximality of the run $(o(k), a(k), b(k), o'(k), a'(k), b'(k))$) that $(a'(k))(p') \sim (b'(k))(p')$, hence $a'(p) \sim b'(p)$.
  - If $o''(p) \neq \mathcal{X}$, then $(o''(k))(p') \neq \mathcal{X}$, hence we have (by maximality of the run $(o(k), a(k), b(k), o'(k), a'(k), b'(k))$) that $(o'(k))(p') \neq \mathcal{X}$, hence $o'(p) \neq \mathcal{X}$. □

6 Case Study: Address Books

We now present a more detailed case study, illustrating how schemas can be used to guide the behavior of our generic synchronizer on trees of realistic complexity. The examples use an address book schema loosely based on the vCard standard [16, 7], which embodies some of the tricky issues that can arise when synchronizing larger structures with varied substructure.

Schemas

We begin with a concrete notion for writing down schemas. Schemas are given by sets of mutually recursive equations of the form `X = S`, where `S` is generated by the following grammar:

$$\texttt{S} ::= \texttt{\{\}} \mid \texttt{n[S]} \mid \texttt{!(F)[S]} \mid \texttt{*(F)[S]} \mid \texttt{S,S} \mid \texttt{S|S}$$

Here `n` ranges over names from $\mathcal{N}$ and `F` ranges over finite sets of names. The first form of schema, `{}`, denotes the singleton set containing only the empty tree; `n[S]` denotes the set of trees with a single child named `n` where the subtree under `n` is in `S`; the wildcard schema `!(F)[S]` denotes the set of trees with *any single child* not in `F`, where the subtree under that child is in `S`; the other wildcard schema, `*(F)[S]` denotes the set of trees with *any number of children* not in `F` where the subtree under each child is in `S`. The set of trees described by $\texttt{S}_1\texttt{|}\texttt{S}_2$ is the union of the sets described by $\texttt{S}_1$ and $\texttt{S}_2$, while $\texttt{S}_1\texttt{,}\texttt{S}_2$ denotes the set of trees that can be "split" into two where one part is in $\texttt{S}_1$ and the other is in $\texttt{S}_2$. Note that, because trees are unordered, the "`,`" operator is commutative; for example, `n[X],m[Y]` and `m[Y],n[X]` are equivalent schemas. We often abbreviate `n[S]|{}` as `n?[S]`, abbreviate `!(∅)[S]` as `![S]`, and abbreviate `*(∅)[S]` as `*[S]`.

All of the schemas we write in this section are path consistent. (This may easily be verified from the fact that, whenever a given name appears twice as a child of a given node, like `m` in `m[X],n[Y]|m[X],o[Z]`, the types of the associated subtrees are textually identical.)

Address Book Schema

The individual address book contacts in our examples have structure similar to the following tree (the notation $[t_1; \ldots; t_n]$, which represents a list encoded as a tree, is explained below):

$$o = \left\{ \begin{array}{l} \texttt{name} \mapsto \left\{ \begin{array}{l} \texttt{first} \mapsto \texttt{Meg} \\ \texttt{other} \mapsto [\texttt{Liz}; \texttt{Jo}] \\ \texttt{last} \mapsto \texttt{Smith} \end{array} \right\} \\ \texttt{email} \mapsto \left\{ \begin{array}{l} \texttt{pref} \mapsto \texttt{msmith@city.edu} \\ \texttt{alts} \mapsto \texttt{meg@smith.com} \end{array} \right\} \\ \texttt{home} \mapsto \texttt{555-6666} \\ \texttt{work} \mapsto \texttt{555-7777} \\ \texttt{org} \mapsto \left\{ \begin{array}{l} \texttt{orgname} \mapsto \texttt{City University} \\ \texttt{orgunit} \mapsto \texttt{Dept of CS} \end{array} \right\} \end{array} \right\}$$

There are two sorts of contacts—"professional" contacts, which contain mandatory work phone and organization entries plus, optionally, a home phone, and "personal" ones, which have a mandatory home phone and, optionally, a work phone and organization information. Contacts are not explicitly tagged with their sort, so some contacts, like the one for `Meg` shown above, have both sorts. Each contact also contains fields representing name and email address information. Both sorts of contacts have natural schemas that reflects their record-like structures with optional fields. Using a union, we can write a schema, `C`, that describes both sorts of contacts.

```
C = name[N],work[V],home?[V],org[O],email[E]
  | name[N],work?[V],home[V],org?[O],email[E]
```

The sub-schemas `V`, `N`, `E`, and `O` are described below.

The trees appearing under the `home` and `work` children are simple values—i.e., trees with a single child leading to the empty tree. The `V` schema, defined as

```
V = ![{}]
```

denotes the set of all such trees.

The `name` edge maps to a tree with a record-like structure containing mandatory `first` and `last` fields and an optional `other` field. The `first` and `last` fields map to values belonging to the `V` schema. The `other` field maps to a list of alternate names such as middle names or nicknames, stored (for the sake of the example) in some particular order. Because our actual trees are unordered, we use a standard "cons cell" representation (as in Lisp) to encode ordered lists. The list $[t_1; \ldots; t_n]$ is encoded as the tree

$$\left\{\begin{array}{l}\texttt{head} \mapsto \texttt{t}_1 \\ \texttt{tail} \mapsto \left\{\ldots \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{t}_n \\ \texttt{tail} \mapsto \texttt{nil}\end{array}\right\}\ldots\right\}\end{array}\right\}$$

Using this representation of lists, the `N` schema is defined straightforwardly as:

```
N = first[V],other?[VL],last[V]
VL = head[V],tail[VL] | nil
```

The `VL` schema describes lists of values (encoded as trees).

The email address data for a contact is either a single value, or a set of addresses with one distinguished "preferred" address. The `E` schema describes these structures using a union of a wildcard to represent single values (which excludes `pref` and `alts` to ensure path consistency) and a record-like structure with fields `pref` and `alts` to represent sets of addresses.

```
E =  !(pref, alts)[{}] | pref[V],alts[VS]
VS = *[{}]
```

The `VS` schema, which describes the trees that may appear under `alts`, denotes bushes with any number of children where each child maps to the empty tree. These bushes are a natural encoding of sets of values as trees.

Finally, organization information is represented by a structure with `orgname` and `orgunit` fields, each leading to a value, as described by this schema:

```
O = orgname[V],orgunit[V]
```

Keys

We can synchronize a whole address book by representing it as a bush with the *key field* for each contact providing access to the contact itself.[4] The key fields, which uniquely identify a contact, are often drawn from some underlying

[4]Note that this is probably not the way that the address book is represented concretely, in an XML file stored on the disk: the concrete representation must be transformed into this form before synchronization. Also, after synchronization, the updated replicas must be transformed back into the appropriate concrete format. The Harmony system includes a domain-specific programming language for writing these bi-directional transformations [12].

database:

$$\left\{\begin{array}{l} \texttt{92373} \mapsto \left\{\begin{array}{l} \texttt{name} \mapsto \left\{\begin{array}{l} \texttt{first} \mapsto \texttt{Megan} \\ \texttt{last} \mapsto \texttt{Smith} \end{array}\right\} \\ \texttt{home} \mapsto \texttt{555-6666} \end{array}\right\} \\ \texttt{92374} \mapsto \left\{\begin{array}{l} \texttt{name} \mapsto \left\{\begin{array}{l} \texttt{first} \mapsto \texttt{Pat} \\ \texttt{last} \mapsto \texttt{Jones} \end{array}\right\} \\ \texttt{home} \mapsto \texttt{555-2222} \end{array}\right\} \\ \texttt{92375} \mapsto \left\{\begin{array}{l} \texttt{name} \mapsto \left\{\begin{array}{l} \texttt{first} \mapsto \texttt{Jill} \\ \texttt{last} \mapsto \texttt{Walters} \end{array}\right\} \\ \texttt{work} \mapsto \texttt{555-1111} \\ \texttt{org} \mapsto \{\ldots\} \end{array}\right\} \end{array}\right\}$$

The effect during synchronization will be that entries from the two replicas with the same UID will be synchronized with each other. Alternatively, if UIDs are not available, we can synthesize a UID by lifting some information out of each record. For example, we might concatenate the `name` information and use it as the top-level key field:

$$\left\{\begin{array}{l} \texttt{Megan:Smith} \mapsto \{\texttt{home} \mapsto \texttt{555-6666}\} \\ \texttt{Pat:Jones} \mapsto \{\texttt{home} \mapsto \texttt{555-2222}\} \\ \texttt{Jill:Walters} \mapsto \left\{\begin{array}{l} \texttt{work} \mapsto \texttt{555-1111} \\ \texttt{org} \mapsto \{\ldots\} \end{array}\right\} \end{array}\right\}$$

The schema for an entire address book represented using external UIDs is:

```
AB = *[C]
```

That is, `AB` denotes the set of trees with any number of children labeled by keys, where each key maps to a contact belonging to `C`. The schema for an entire address book represented using synthesized keys is:

```
AB = *[C']
```

where `C'` is a schema just like `C` that does not include fields for `name` data, which is instead lifted out above each entry as a key.

For the rest of the case study, we will focus our attention on the synchronization of individual address book entries, because this is where the schema plays the most interesting role.

The Need For Schemas

To illustrate how and where schema conflicts can occur, let us see what can go wrong when *no* schema information is used. We consider four runs of the synchronizer using the all-inclusive schema, each showing a different way in which schema-ignorant synchronization can produce mangled results:

```
Any = *[Any]
```

In each case, the archive, o, is the tree shown above.

Suppose, first, that the a replica is obtained by deleting the `work` and `org` children, making the entry personal, and that the b replica is obtained by deleting the `home` child, making the entry professional:

$$a = \left\{\begin{array}{l} \texttt{name} \mapsto \left\{\begin{array}{l} \texttt{first} \mapsto \texttt{Meg} \\ \texttt{other} \mapsto [\texttt{Liz; Jo}] \\ \texttt{last} \mapsto \texttt{Smith} \end{array}\right\} \\ \texttt{email} \mapsto \left\{\begin{array}{l} \texttt{pref} \mapsto \texttt{msmith@city.edu} \\ \texttt{alts} \mapsto \texttt{meg@smith.com} \end{array}\right\} \\ \texttt{home} \mapsto \texttt{555-6666} \end{array}\right\} \qquad b = \left\{\begin{array}{l} \texttt{name} \mapsto \left\{\begin{array}{l} \texttt{first} \mapsto \texttt{Meg} \\ \texttt{other} \mapsto [\texttt{Liz; Jo}] \\ \texttt{last} \mapsto \texttt{Smith} \end{array}\right\} \\ \texttt{email} \mapsto \left\{\begin{array}{l} \texttt{pref} \mapsto \texttt{msmith@city.edu} \\ \texttt{alts} \mapsto \texttt{meg@smith.com} \end{array}\right\} \\ \texttt{work} \mapsto \texttt{555-7777} \\ \texttt{org} \mapsto \left\{\begin{array}{l} \texttt{orgname} \mapsto \texttt{City University} \\ \texttt{orgunit} \mapsto \texttt{Dept of CS} \end{array}\right\} \end{array}\right\}$$

Although a and b are both valid address book contacts, the trees that result from synchronizing them with respect to the Any schema are not, since they have the structure neither of personal nor of professional contacts:

$$a' = b' = \left\{ \begin{array}{l} \texttt{name} \mapsto \left\{ \begin{array}{l} \texttt{first} \mapsto \texttt{Meg} \\ \texttt{other} \mapsto [\texttt{Liz};\ \texttt{Jo}] \\ \texttt{last} \mapsto \texttt{Smith} \end{array} \right\} \\ \texttt{email} \mapsto \left\{ \begin{array}{l} \texttt{pref} \mapsto \texttt{msmith@city.edu} \\ \texttt{alts} \mapsto \texttt{meg@smith.com} \end{array} \right\} \end{array} \right\}$$

Next, suppose that the replicas are obtained by updating the trees along the path name/first, replacing Meg with Maggie in a and Megan in b. (From now on, for the sake of brevity we only show the parts of the tree that are different from o and elide the rest.)

$$\begin{array}{rcl} o(\texttt{name/first}) & = & \texttt{Meg} \\ a(\texttt{name/first}) & = & \texttt{Maggie} \\ b(\texttt{name/first}) & = & \texttt{Megan} \end{array}$$

Synchronizing with respect to the Any schema yields results where *both* names appear under first:

$$a'(\texttt{name/first}) = b'(\texttt{name/first}) = \left\{ \begin{array}{l} \texttt{Maggie} \\ \texttt{Megan} \end{array} \right\}$$

These results are ill-formed because they do not belong to the V schema, which describes trees that have a *single* child.

For the next example, consider updates to the email information where the a replica replaces the set of addresses in o with a single address, and b updates both pref and alts children in b:

$$\begin{array}{rcl} o(\texttt{email}) & = & \left\{ \begin{array}{l} \texttt{pref} \mapsto \texttt{msmith@city.edu} \\ \texttt{alts} \mapsto \texttt{meg@smith.com} \end{array} \right\} \\ a(\texttt{email}) & = & \left\{ \texttt{meg@smith.com} \right\} \\ b(\texttt{email}) & = & \left\{ \begin{array}{l} \texttt{pref} \mapsto \texttt{meg.smith@cs.city.edu} \\ \texttt{alts} \mapsto \texttt{msmith@city.edu} \end{array} \right\} \end{array}$$

Synchronizing these trees with respect to Any propagates the addition of the edge labeled meg@smith.com from a to b' and yields conflicts on both pref and alts children, since both have been deleted in a but modified in b. The results after synchronizing are thus:

$$\begin{array}{rcl} a'(\texttt{email}) & = & \texttt{meg@smith.com} \\ b'(\texttt{email}) & = & \left\{ \begin{array}{l} \texttt{meg@smith.com} \\ \texttt{pref} \mapsto \texttt{meg.smith@cs.city.edu} \\ \texttt{alts} \mapsto \texttt{msmith@city.edu} \end{array} \right\} \end{array}$$

The second result, b', is ill-formed because it contains three children, whereas all the trees in the email schema E have either one or two children.

As a final example, consider changes to the list of names stored at the path name/other. Suppose that a removes both Liz and Jo, but b only removes Jo:

$$\begin{array}{rcl} o(\texttt{name/other}) & = & [\texttt{Liz};\ \texttt{Jo}] \\ a(\texttt{name/other}) & = & [] \\ b(\texttt{name/other}) & = & [\texttt{Liz}] \end{array}$$

To see what goes wrong when we synchronize, let us rewrite each list explicitly as a tree:

$$\begin{array}{rcl} o(\texttt{name/other}) & = & \left\{ \begin{array}{l} \texttt{head} \mapsto \texttt{Liz} \\ \texttt{tail} \mapsto \left\{ \begin{array}{l} \texttt{head} \mapsto \texttt{Jo} \\ \texttt{tail} \mapsto \texttt{nil} \end{array} \right\} \end{array} \right\} \\ a(\texttt{name/other}) & = & \texttt{nil} \\ b(\texttt{name/other}) & = & \left\{ \begin{array}{l} \texttt{head} \mapsto \texttt{Liz} \\ \texttt{tail} \mapsto \texttt{nil} \end{array} \right\} \end{array}$$

Comparing the a replica to o, both head and tail are deleted and nil is newly added. Examining the b replica, the tree under head is identical to corresponding tree in o but deleted from a. The tree under tail is different from o

but deleted from a. Finally, the edge `nil` is not present in o but has been added in a. Collecting all of these changes, the algorithm yields these results:

$$\begin{array}{rcl} a'(\texttt{name/other}) & = & \texttt{nil} \\ b'(\texttt{name/other}) & = & \left\{\begin{array}{l}\texttt{tail} \mapsto \texttt{nil} \\ \texttt{nil}\end{array}\right\} \end{array}$$

Here again, the second result, b', is ill-formed: it has children `tail` and `nil`, which is not a valid encoding of any list.

Situations like these—invalid record structures, multiple children under where a single value is expected, and mangled list structures—provided the initial motivation for equipping a straightforward "tree-merging" synchronization algorithm with schema information. Fortunately, in all of these examples, the step that breaks the structural invariant can be detected by a simple, local, domain test. In the first example, where the algorithm removed the `home`, `work`, and `org` children, the algorithm tests if $\{\texttt{name}, \texttt{email}\}$ is in $doms(\texttt{C})$. Similarly, in the second example, where both replicas changed the `first` name to a different value, the algorithm tests if $\{\texttt{Maggie}, \texttt{Megan}\}$ is in $doms(\texttt{V})$. In the example involving the tree under `email`, the algorithm tests if the domain $\{\texttt{meg@smith.com}, \texttt{pref}, \texttt{alts}\}$ is in $doms(\texttt{E})$. Finally, in the example where both replicas updated the list of `other` names, it tests whether $\{\texttt{tail}, \texttt{nil}\}$ is in $doms(\texttt{VL})$. All of these local tests fail and so the synchronizer halts with a schema domain conflict at the appropriate path in each case, ensuring that the results are valid according to the schema.

These examples illustrate how schemas can be used to control the behavior of a synchronization algorithm in many simple situations by halting with a schema domain conflicts when a well-formed result cannot be locally constructed at a node. In the remainder of the section, we explore the strengths and weaknesses of this local algorithm by examining its behavior in more detail on the different structures that appear in the address book contact schema. Appendix A discusses some additional structures including tuples, relations, and XML trees.

Synchronizing Values

The simplest structures in our address books, values, are represented as trees having a single child that, in turn, leads to an empty tree; they are described by the schema `![{}]`. When we synchronize two non-missing trees with respect to this schema, there are only a three possible scenarios. First, if either of a or b is identical to o, as in this example

$$\begin{array}{rcl} o & = & \texttt{Meg} \\ a & = & \texttt{Maggie} \\ b & = & \texttt{Meg} \end{array}$$

then the algorithm set the results equal to the other replica:

$$a = b = \texttt{Maggie}$$

Alternatively, if a and b are identical to each other but different to o, as in

$$\begin{array}{rcl} o & = & \texttt{Meg} \\ a & = & \texttt{Maggie} \\ b & = & \texttt{Maggie} \end{array}$$

then the algorithm preserves the equality:

$$a = b = \texttt{Maggie}$$

Finally, if a and b are both different from o and each other, as in

$$\begin{array}{rcl} o & = & \texttt{Meg} \\ a & = & \texttt{Maggie} \\ b & = & \texttt{Megan} \end{array}$$

then the algorithm reaches a schema domain conflict, setting $o' = \mathcal{X}$, $a' = a$ and $b' = b$. These behaviors follow from the alignment mechanism employed in the algorithm—children with the same name are identified across replicas and

so identical values are aligned with each other and distinct values synchronized separately. In the first two scenarios, the differences in a and b (with respect to o) can be assembled into a value; in the third scenario, they cannot. Therefore, on values, the schema-aware synchronization algorithm enforces *atomic* changes, propagating updates from one side to the other only if o and either a or b are identical and otherwise leaving the replicas unchanged and signalling a schema domain conflict.

Synchronizing Sets

A set of atoms is represented as a bush, where each child in the bush maps to the empty tree; this structure is described by the schema `*[{}]`. Interestingly, when synchronizing two sets of atoms, the synchronization algorithm *never* reaches a schema conflict; it can always produce a well-typed result by combining arbitrary additions and deletions of atoms from a and b. For example, given these three sets:

$$\begin{array}{rcl} o & = & \{\texttt{meg@smith.com}\} \\ a & = & \left\{\begin{array}{l}\texttt{msmith@city.edu} \\ \texttt{meg.smith@cs.city.edu}\end{array}\right\} \\ b & = & \left\{\begin{array}{l}\texttt{meg@smith.com} \\ \texttt{meg.smith@cs.city.edu}\end{array}\right\} \end{array}$$

The synchronizer propagates the deletion of `meg@smith.com` and the addition of two new children,

`msmith@city.edu.com` and `meg.smith@cs.city.edu`

producing results:

$$a' = b' = \left\{\begin{array}{l}\texttt{msmith@city.edu} \\ \texttt{meg.smith@cs.city.edu}\end{array}\right\}$$

as expected.

Synchronizing Records

Record structures are used to organize data in several different parts of the address book schema. The simplest records, as in the schema representing organization data, `orgname[V],orgunit[V]`, have a fixed set of mandatory fields. Given two encodings of records in the same schema, the synchronizer aligns the common fields, which are *all* guaranteed to be present, and synchronizes the nested data one level down. Hence, we never reach a schema domain conflict at the root node. Other kinds of records, which we call *sparse records*, allow some variation in the names of their immediate children. For example, the contact schema uses a sparse record to represent the structure of each entry where some fields, like `org`, may be optionally present or missing (depending on the presence of other fields such as `work`):

```
  name[N],work[V],home?[V],org[O],email[E]
| name[N],work?[V],home[V],org?[O],email[E]
```

As we saw in the preceeding section, there are runs of the synchronizer that cannot propagate all of the changes from a to b while preserving the structure expressed by some sparse record schema. In these situations, the synchronizer must yield a schema conflict.

Synchronizing Lists

The address book schema uses lists of values to represent the ordered collection of optional `other` names for a contact. We included this example to show how our synchronization algorithm behaves when applied to structures that are a little beyond its intended scope—i.e., mostly consisting of unordered or rigidly ordered data, but also containing small amounts of list-structured data.

Lists present special challenges, because we would ideally expect the algorithm to detect updates both to individual elements and to the relative position of elements; however, our local alignment strategy matches up and recursively synchronizes list elements by their *absolute* positions, leading to surprising results on some inputs. We illustrate the

problem and propose a more sophisticated encoding of lists that minimizes (but does not eliminate) the chances of confusion.

Let us first consider runs of the synchronizer where changes in one replica can be successfully propagated to the other. If either replica is identical to the archive, then the algorithm trivially copies all of the changes from the other replica. Or, if both replicas are not equal to the archive but each replica modifies a disjoint subset of the elements of the list (and leaves the spine of the list intact), then the synchronizer can merge the changes successfully. At each step in a recursive tree walk only one of the elements will have changed, as in the following example. (In this section, we consider lists of values, and write out the low-level tree representation for each tree.)

$$\begin{aligned} o &= \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Liz} \\ \texttt{tail} \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Jo} \\ \texttt{tail} \mapsto \texttt{nil}\end{array}\right\}\end{array}\right\} \\ a &= \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Elizabeth} \\ \texttt{tail} \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Jo} \\ \texttt{tail} \mapsto \texttt{nil}\end{array}\right\}\end{array}\right\} \\ b &= \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Liz} \\ \texttt{tail} \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Joanna} \\ \texttt{tail} \mapsto \texttt{nil}\end{array}\right\}\end{array}\right\} \end{aligned}$$

The changes under `head` are propagated from a to b', and the changes to `tail` from b to a', yielding results:

$$a' = b' = \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Elizabeth} \\ \texttt{tail} \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Joanna} \\ \texttt{tail} \mapsto \texttt{nil}\end{array}\right\}\end{array}\right\}$$

There are some inputs, however, where synchronizing lists using the local alignment strategy and simple cons cell encoding produces strange results. Consider a run of the synchronizer on the following trees (where o is the same tree as above):

$$\begin{aligned} a &= \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Jo} \\ \texttt{tail} \mapsto \texttt{nil}\end{array}\right\} \\ b &= \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Liz} \\ \texttt{tail} \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Joanna} \\ \texttt{tail} \mapsto \texttt{nil}\end{array}\right\}\end{array}\right\} \end{aligned}$$

Considering the changes that were made to each list from a high-level—a removed the head and b renamed the second element—the result calculated for b' is somewhat surprising:

$$b' = \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Jo} \\ \texttt{tail} \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{Joanna} \\ \texttt{tail} \mapsto \texttt{nil}\end{array}\right\}\end{array}\right\}$$

The algorithm does not recognize that `Jo` and `Joanna` should be aligned (because `Joanna` was obtained by renaming `Jo`). Instead, it aligns pieces of the list by absolute position, matching `Jo` with `Liz` and `nil` with [`Joanna`].

It is not surprising that our algorithm doesn't have intuitive behavior when its inputs are lists. In general, detecting changes in relative position in a list requires global reasoning but our algorithm is essentially local. We reluctantly conclude that in its current form, our algorithm is not well-suited to this form of synchronization.

In order to avoid these problematic cases, we can use an alternative schema, which we call the *keyed list schema*, for lists whose relative order matters. Rather than embedding the elements under a spine of cons cells, one can include the value at each position as part of the cons cell encoding. For example, in the extended encoding, the list o from above is represented as the tree:

$$o = \left\{\texttt{Liz} \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \{\} \\ \texttt{tail} \mapsto \left\{\texttt{Jo} \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \{\} \\ \texttt{tail} \mapsto \texttt{nil}\end{array}\right\}\right\}\end{array}\right\}\right\}$$

(For keyed lists of values, we could drop the child `head`, which always maps to the empty tree. However, the keyed list schema can also be used to form keyed lists of arbitrary trees, not just values. For trees with structure richer than values, we would use some key field at the top of each cons cell and store the rest of the tree under `head`.)

The schema for keyed lists of values is straightforward:

```
KVL = !(nil)[head[],tail[KVL]] | nil
```

During synchronization, elements of the list are identified by the the value above each cons cell, and synchronization proceeds until a trivial case applies (unchanged replica or identical replicas), or when the two replicas disagree on the domain of one element, resulting in a schema domain conflict. In both of the previous examples, the algorithm terminates with a schema domain conflict at the root. This workaround introduces extra schema domain conflicts but avoids the unintuitive behavior that appears when the simpler encoding is used. However, because the keyed list encoding requires identifying suitable key fields every time a list structure is needed, we do not currently use it extensively in our implementation.

Summary

The examples in this section demonstrate that schemas are a valuable addition to a synchronization algorithm, producing safe, well-formed results in many situations where a simpler schema-blind algorithm would yield ill-formed results. Our purely local algorithm works well with rigidly structured data (like values and records) and unstructured data (like sets of values), but has limited utility when used with ordered and semi-structured data (like lists and documents). In the future, we hope to extend our algorithm to better handle ordered data.

7 Design Issues and Related Work

The Harmony framework combines a core synchronization component with a view update component for dealing with heterogeneity. The view update language is described in [12], which also contains a detailed discussion of related work on this topic. Here, we concentrate on related work on optimistic replication and synchronization.

The reader is directed to an excellent article by Saito and Shapiro [37] surveying the area of optimistic replication. In the taxonomy of the survey, Harmony is a multi-master state-transfer system, recognizing sub-objects and manually resolving conflicts. However, some important distinctions raised in this paper are not adequately covered in the aforementioned taxonomy.

In particular, Harmony is a generic synchronization framework, with a goal of supporting reconciliation even of instances of distinct off-the-shelf applications, running on heterogeneous platforms. This goal drives us to an extremely loose coupling between the synchronizer and the applications it is synchronizing, which in turn motivates our use of the state-transfer approach. Our goal of synchronizing distinct applications, with different concrete representations of the shared state, drives us to use lenses to transform our concrete views to abstract trees that are instances of a shared, per-application, schema. Our desire to use only mechanisms that are simple to understand and easy to formalize has led us to experiment with pushing almost the entire burden of aligning substructures within replicas to the lenses, which allows us to have a single, simple, generic algorithm for performing synchronization. Independently, we strive for predictable behavior even when running unsupervised, which leads us to value persistence over convergence. Both the heterogeneity of our replicas and the state-based approach of our reconciler have led us into under-investigated areas in the design space of optimistic reconciliation.

Loosely vs. Tightly Coupled Reconcilers

Harmony is a generic framework centered around a loose coupling between the reconciler and the application whose state is being replicated. The goal of loose coupling led us to use a state-based approach to reconciliation, rather than an operation-based approach. In general, reconcilers cannot expect to be able to know the operation history if they are to synchronize off-the-shelf, proprietary applications that have not been constructed to be "synchronization aware." In addition, the behavior of a state-based reconciler is much simpler, which makes it easier for users to predict the outcome of reconciliation.

However, there are also drawbacks to the state-based approach when compared to operation-based architectures. State-based architectures have less information available at synchronization time; they cannot exploit knowledge of temporal sequencing that is available in operation logs. The operation logs can sometimes determine that two modifications are not in conflict, if one is in the operation history of the other. Further, in a tightly coupled architecture,

the designer can choose to expose operations that encode the high-level application semantics. The synchronizer will then manipulate operations that are close to the actual user operations. This can preserve a primitive type of atomicity: treating user-level operations as primitives makes it more likely from the perspective of the user that, even under conflict, the system will be in a "reasonable" state.

The distinction between state-based and operation-based synchronizers is not black and white: various hybrids are possible. For example, we can build a state-based system with an operation-based core by comparing previous and current states to obtain a hypothetical (typically, minimal) sequence of operations. But this involves complex heuristics, which can conflict with our goal of presenting predictable behavior to the user. Similarly, some loosely-coupled systems can build an operation-based system with a state-transfer core by using an operation log in order to determine what part of the state to transfer.

One seemingly novel feature of Harmony is that we *transform* our data structures several times in the course of reconciliation. However, a form of tranformation also occurs in some operation-based reconcilers. Operation-based reconcilers attempt to merge their log of operations in such a way that, after quiescence, if each replica applies its merged log to the last synchronized state, then all replicas share a uniform state. There are, broadly speaking, three alternatives to merging logs: (1) reorder operations on all replicas to achieve an identical schedule (c.f. Bayou [9]), (2) partially reorder operations, exploiting semantic knowledge to leave equivalent sequences unordered (c.f. IceCube [18]), or (3) perform no reordering, but transform the operations themselves, so that the different schedules on each different replica all have a uniform result (c.f. [25]). The best schedule is one in which conflicts between operations are minimized.

The third approach mentioned above, called *operational transformation*, performs transformations as does Harmony. However, the nature of the transformations are substantially different: Systems that use operational transformation (e.g. [6, 28, 39, 24, 17, 25]) transform operations; Harmony transforms local data structures. operational transformation systems transform concurrent operations to reach convergence, Harmony transforms heterogeneous concrete formats to align them. We will return to the relationship between operational transformation and Harmony when we discuss convergence.

Reconciliation Systems

Many other systems support optimistic replicas. Few support heterogeneous replicas, or do much schema-based pre-alignment, but many have other similarities to our work. Harmony is a generic state-based reconciler that is parameterized by the lenses that transform each concrete representation to a shared abstract view (and back again). IceCube [31, 18] is a generic operation-based reconciler that is parameterized over a specific algebra of operations appropriate to the application data being synchronized and by a set of syntactic/static and semantic/dynamic ordering constraints on these operations. Molli et al [24, 17, 25], have also implemented a generic operation-based reconciler, using the technique of operational transformation. Their synchronizer is parameterized on transformation functions for all operations, which must obey certain conditions. Like us, they formally specify the behavior of their system.

Bengal [10] is operation-based only in the sense that it traps each operation and records it in a log, but in fact it uses the operation log strictly as an optimization to avoid scanning the entire replica during update detection. Like Harmony, Bengal is a loosely-coupled synchronizer. It exploits exported OLE/COM hooks, and can extend any commercial database system that uses OLE/COM hooks to support optimistic replication. However, it is not generic because it only supports databases, it is not heterogeneous because reconciliation can only occur between replicas of the same database, and it requires users to write *conflict resolvers* if they want to avoid manually resolving conflicts.

FCDP [19] is a generic, state-based reconciler parameterized by ad-hoc translations from heterogeneous concrete representations to XML and back again. There is no formal specification and reconciliation takes place at "synchronization servers" that are assumed to be more powerful machines permanently connected to the network. Broadly speaking, FCDP can be considered an instance of the Harmony architecture—but without the formal underpinnings. FCDP is less generic (our lens language makes it easier to extend Harmony to new applications), but it is better able to deal with certain edits to documents than Harmony. However, FCDP is more rigid than Harmony in its treatment of ordered lists. FCDP fixes a specific semantics for ordered lists—particularly suited for document editing. This interpretation may sometimes be problematic, as we saw in Section 6.

File system synchronizers (such as [38, 27, 14, 35, 3, 33]) and PDA synchronizers (such as Palm's HotSync), are

not generic, but they do generally share Harmony's state-based approach. An interesting exception is DARCS [36], a hybrid state-/operation-based revision control system built on a "theory of patches."

Convergence and Partial Convergence

Harmony, unlike many reconcilers, does not guarantee convergence in the case of conflicts. A successful run of a reconciler aims to converge; that is, all of the replicas in the system should eventually reach a uniform state. In the case of conflicts, reconcilers can choose one of three broad strategies.

- They can settle all conflicts by fiat, for example by choosing the variant with the latest modification time and discarding the other.
- They can converge without resolving the conflict. In other words, they can keep enough information to record *both* conflicting updates, and converge to a single state in which both replicas include the union of the conflicting updates.
- They can choose to diverge. They can maintain the conflicting updates locally, only, and not converge until the conflicts are (manually) resolved.

The first option is clearly undesirable, and most modern reconcilers will not simply discard updates (they follow a "no lost updates" policy). We note that Harmony, unlike many other reconcilers, chooses the third option (divergence) over the second option (unconditional convergence). Systems such as Ficus [34], Rumor [14], Clique [35], Bengal [10], and TAL/S5 [24, 17, 25] converge by making additional copies of primitive objects that conflict and renaming one of the copies. CVS embeds markers in the bodies of files where conflicts occurred. In contrast, systems such as Harmony and Ice-cube [18] will not reconcile objects affected by conflicting updates. Systems that allow reconciliation to end with divergent replicas have a further choice. They must choose whether to leave the replicas completely untouched by reconciliation, or to try to achieve *partial convergence*. Harmony aims for partial convergence. In Section 5 we show that Harmony is a *maximal synchronizer*, propagating as many changes as possible without losing any updates.

In practice, the difference between systems that allow divergence and systems that guarantee convergence does not seem fundamental. However, we find advantages to Harmony's choice of persistence over convergence from an engineering point of view.

First, by maintaining divergent replicas, it is easier to make unsupervised reconciliations safer. (Unsupervised reconciliations seem extremely desirable from the point of view of system administration. Automation by running nightly reconciliation scripts as well as triggering reconciliation on dis/connection from/to networks seems required in order to make administration manageable.) Divergent systems are more likely to allow users to proceed with their work (the set of replicas may be globally inconsistent, but it is more likely that each replica is locally consistent). Convergent systems are more likely to force a user to resolve a conflict after a *remote* user initiated a synchronization attempt. For example, consider conflicting updates to a file with strict syntax requirements (e.g. LaTeX or C). The convergent system's attempt to record both updates may result in a file that causes subsequent processing to fail.

Second, divergent systems are less likely to hide conflicts for long periods of time. Divergent systems will continue to remind the users of the conflict at every synchronization attempt until the conflict is resolved. (Partial convergence will ensure that the set of such synchronization failures is as small as possible.) Convergent systems will reconcile without problem after a single completed synchronization attempt, even if conflicts persist, because the replicas will be identical. Further, convergent systems must take care that the conflicting updates are marked by out-of-band markers that truly cannot appear in the normal course of system operation, and that cannot disappear without the underlying conflict simultaneously being resolved.

Finally, a primary goal of Harmony is a clear specification—both formal and intuitive—of its behavior. If we claim that Harmony *always* converges then we must prove that it converges even if a synchronization attempt is aborted or preempted before completion. This seems difficult to guarantee, and harder to prove. If we claim that it converges in only *some* cases, but not in others, then we must carefully identify the cases in which it converges and which it does not. Such a complex specification seems likely to be both error-prone and non-intuitive.

Like Harmony, the synchronizer of Molli et al [24, 17, 25] uses formal specifications to ensure safety, but unlike Harmony it chooses convergence over persistence of user changes. The advantage of persistence over convergence is more compelling for Harmony than for Molli's system, because of our interest in unsupervised runs. As such, it is important to specify to users precisely when Harmony will detect conflicts. Molli's synchronizer is satisfied with recording multiple conflicting versions in the reconciled replicas, and restricts its specification to the correctness of its transformation functions.

At first glance, this may seem preferable to our approach, if one believes that conflicts are far rarer in operational transformation systems than in Harmony. However, unique, unambiguous operational transforms may not always exist, increasing the likelihood of conflicts. Operational transforms resolve conflicting schedules by transforming local operations to undo the local operation, then perform the remote operation, and finally redo the local operation. Understanding the correct behavior of "undo" in a collaborative environment is a prerequisite to the correct behavior of operational transformation. Munson and Dewan [26] note that group "undo" may remove the need for a merge capability in optimistic replication. Prakash and Knister [30] provide formal properties that individual primitive operations in a system must satisfy in order to be "undo"able in a groupware setting. Abowd and Dix [2] formally describe the *desired* behavior of undo (and hence of conflict resolution) in "groupware", and identify cases in which undo is fundamentally ambiguous. In such ambiguous cases—even if the primitive operations are defined to have unique undo functions—the user's intention cannot be preserved and it is preferable to report conflict than to lose a user's modification. Lechtenborger [20] shows that update operations are undoable by other update operations precisely in the case that constant complement translators exist.

Heterogeneous Replicas

The Harmony system, with its goal of reconciling heterogeneous data sources, has strong connections with the area of data integration.

Answering queries from heterogeneous data sources is a well-studied area in the context of data integration [11, 1, 15, 40]. If we consider the (non-trivial) problem of augmenting a data integration system with view update (another well-studied area—see [12] for a survey), then the result can be used to implement an optimistic replication system that can reconcile conflicts between heterogeneous data sources[5]. However, to the best of our knowledge, no generic synchronizer other than Harmony supports reconciliation over truly heterogeneous replicas. FCDP [19] is designed to be generic, but the genericity is limited to using XML as the internal representation, and currently only reconciles documents. Some file synchronizers do support diversity in small ways. For example, file synchronizers often grapple with different representations of file names and properties when reconciling between two different system types. Some map between length-limited and/or case insensitive names and their less restrictive counterparts (c.f. [3, 35]). Others map complex file attributes (e.g. the Macintosh resource fork) into directories, rather than files, on the remote replicas.

Harmony's emphasis on schema-based pre-alignment is influenced by examples we have found in the context of data integration where heterogeneity is a primary concern. Alignment, in the form of schema-mapping, has been frequently used to good effect (c.f. [32, 23, 4, 8, 22]). The goal of alignment, there, is to construct views over heterogeneous data, much as we transform concrete views into abstract views with a shared schema to make alignment trivial for the reconciler.

Some synchronizers differ mainly in their treatment of alignment strategy. For example, in terms of features, the main difference between Unison [3, 29] (which has almost trivial alignment) and CVS, is the comparative alignment strategy (based on the standard Unix tool `diff3`) used by CVS. At this stage, Harmony's core synchronization algorithm is deliberately simplistic, particularly with respect to ordered data. As we develop an understanding of how to integrate more sophisticated alignment algorithms in a generic and principled way, we hope to incorporate them into Harmony. Of particular interest are diff3 and its XML based descendants, such as Lindholm's 3DM [21], the work of Chawathe et al [5], and FCDP [19].

[5]The inverse does not follow. Harmony cannot be used to both solve the general view update problem and support general data integration. Harmony addresses only a subset of the view-update problem that we found necessary to support reconciliation. Similarly, it can integrate concrete views only when the common abstract schema and the lenses that transform views from concrete to abstract, and back again, obey closure properties dictated by our synchronization algorithm.

8 Future Work

In the longer term, a number of directions warrant further investigation.

First, the two-replica-plus-archive algorithm and specification that we have given here should be extended to handle multiple replicas. This extension raises some interesting puzzles concerning the handling of the case where the replicas are different from the archive but equal to each other. Our work toward a design for this extension is reported in [13].

Second, we would like to combine the core features of Harmony with a more sophisticated treatment of ordered structures, as found, for example, in Lindholm's 3DM [21], the work of Chawathe et al [5], and FCDP [19]. Similarly, although the Harmony framework has been designed with unordered tree synchronization in mind, it may be generalizable to richer structures such as DAGs.

Acknowledgments

The Harmony project was begun in collaboration with Zhe Yang; Zhe contributed numerous insights whose genetic material can be found (generally in much-recombined form) in this paper. Trevor Jim provided the initial push to start the project by observing that the next step beyond the Unison file synchronizer (of which Trevor was a co-designer) would be synchronizing XML. Jonathan Moore, Owen Gunden, and Malo Denielou have collaborated with us on many aspects of the Harmony design and implementation. Ongoing work with Sanjeev Khanna and Keshav Kunal on extensions of Harmony's synchronization algorithm has deepened our understanding of the core mechanisms presented here. Conversations with Martin Hofmann, Zack Ives, Nitin Khandelwal, William Lovas, Kate Moore, Cyrus Najmabadi, Stephen Tse, and Steve Zdancewic also helped us sharpen our ideas.

The Harmony project is supported by the National Science Foundation under grant ITR-0113226, *Principles and Practice of Synchronization*. Nathan Foster is also supported by an NSF Graduate Research Fellowship.

References

[1] S. Abiteboul. Querying semi-structured data. In *ICDT*, pages 1–18, 1997.

[2] G. D. Abowd and A. J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

[3] 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.

[4] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *ICDT'99*, 1999.

[5] S. S. Chawathe, A. Rajamaran, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on the management of Data*, pages 493–504, Montreal, Quebec, 1996.

[6] G. V. Cormack. Brief abstract: A calculus for concurrent update. In *Proceedings of the 14th Symposium on Principles of Distributed Computing (PODC '95)*, page 269, August 1995. Ottawa, Canada.

[7] F. Dawson and T. Howes. RFC 2426: vCard MIME directory profile, Sept. 1998.

[8] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, 2001.

[9] W. K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the Tenth ACM Symposium on User Interface Software and Technology (UIST)*, October 1997.

[10] T. Ekenstam, C. Matheny, P. L. Reiher, and G. J. Popek. The Bengal database replication system. *Distributed and Parallel Databases*, 9(3):187–210, 2001.

[11] D. Florescu, A. Y. Levy, and A. O. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[12] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, 2005. Extended version available as University of Pennsylvania technical report MS-CIS-03-08. Earlier version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

[13] M. B. Greenwald, S. Khanna, K. Kunal, B. C. Pierce, and A. Schmitt. Agreement is quicker than domination: Conflict resolution for optimistically replicated data. Submitted for publication; available electronically, 2005.

[14] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. pages 254–265, 1998.

[15] A. Y. Halevy. Theory of answering queries using views. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(4):40–47, 2000.

[16] T. Howes, M. Smith, and F. Dawson. RFC 2425: A MIME content-type for directory information, Sept. 1998.

[17] 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.

[18] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of diverging replicas. In *proceedings of the 20th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '01)*, Aug. 26-29 2001. Newport, Rhode Island.

[19] M. Lanham, A. Kang, J. Hammer, A. Helal, and J. Wilson. Format-independent change detection and propoagation in support of mobile computing. In *Proceedings of the XVII Symposium on Databases (SBBD 2002)*, pages 27–41, October 14-17 2002. Gramado, Brazil.

[20] J. Lechtenbörger. The impact of the constant complement approach towards view updating. In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 49–55. ACM, June 9–12 2003. San Diego, CA.

[21] T. Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *Proceedings of MobiDE '03*, pages 93–97, September 19 2003. San Diego, CA.

[22] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *The VLDB Journal*, pages 49–58, 2001.

[23] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB'98*, 1998.

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

[25] 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.

[26] J. P. Munson and P. Dewan. A flexible object merging framework. In *1994 ACM Conference on Computer Supported Cooperative Work*, pages 231–242, 1994.

[27] 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.

[28] C. Palmer and G. V. Cormack. Operation transforms for a distributed shared spreadsheet. In *Conference on Computer-supported cooperative work (CSCW '98)*, pages 69–78, November 1998. Seattle, WA.

[29] B. C. Pierce and J. Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[30] A. Prakash and M. J. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295–330, 1994.

[31] N. Preguia, M. Shapiro, and C. Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, May 2002.

[32] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[33] 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.

[34] 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.

[35] 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.

[36] D. Roundy. Darcs revision control system, 2004. http://www.abridgegame.org/darcs/.

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

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

[39] C. Sun and C. S. Ellis. Operational transform in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work (CSCW '98)*, pages 59–68, 1998. Seattle, Wash.

[40] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD Conference*, 2001.

K	A	B
k_1	a_1	b_1
k_2	a_2	b_2

can be represented as: $\left\{ k_1 \mapsto \begin{Bmatrix} A \mapsto a_1 \\ B \mapsto b_1 \end{Bmatrix} \; k_2 \mapsto \begin{Bmatrix} A \mapsto a_2 \\ B \mapsto b_2 \end{Bmatrix} \right\}$

Figure 3: Representing relations as trees

A Other Encodings

Here we record some additional encodings that we have developed for tuples, relations, and XML structures.

Tuples

We can encode fixed-width tuples as record-like structures whose labels are in the set $1 \ldots n$. Hence the tuple (t_1, t_2, t_3) may be encoded as the tree $\{\texttt{1} \mapsto t_1; \texttt{2} \mapsto t_2; \texttt{3} \mapsto t_3\}$. The schema follows straightforwardly from this encoding; for the above triple, it is `1[T1],2[T2],3[T3]`. Since each tree in a tuple schema has every child in $1 \ldots n$, the synchronizer aligns elements at the same positions in the tuple and never produces a schema domain conflict at the root.

Relations

Relations, and other data structures consisting of a set of structured elements, present interesting alignment challenges. Considering a relation as a set of tuples, one needs to identify which tuple of the archive, o, should be associated to which tuple of each replica, a and b. Consider for instance the following relations:

$o =$

Pat	333-4444
Chris	888-9999

$a =$

Pat	111-2222
Chris	888-9999

$b =$

Pat	123-4567
Jo	888-9999

If one chooses a schema where tuples are unstructured values, encoding $(\texttt{Pat}, \texttt{333-4444})$ as a single string such as `Pat:333-4444` (using `:` as a separator so that it is easy to extract the components later),

$$\begin{array}{rcl} o & = & \{\texttt{Pat:333-4444 Chris:888-9999}\} \\ a & = & \{\texttt{Pat:111-2222 Chris:888-9999}\} \\ b & = & \{\texttt{Pat:123-4567 Jo:888-9999}\} \end{array}$$

then the result of synchronization, using the schema for sets, `*[{}]`, is the tree

$$\{\texttt{Pat:111-2222}, \texttt{Pat:123-4567}, \texttt{Jo:888-9999}.\}$$

The surprising duplication of the entry for Pat results from the fact that the tuple was considered unstructured, hence `Pat:111-2222` and `Pat:123-4567` are treated as two independent, non-conflicting additions.

A better way to encode this relation is to choose one attribute that is a key and encode a relation as a tree whose children are the key values pointing to a record containing the other attributes, as depicted in Figure 3. This has the effect of aligning identical parts by keys, which can be chosen in an application-appropriate way.

A satisfying representation for our simple example would thus be (e.g., for o):

$$\left\{ \begin{array}{l} \texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{333-4444}\}\} \\ \texttt{Chris} \mapsto \{\texttt{Phone} \mapsto \{\texttt{888-9999}\}\} \end{array} \right\}$$

This representation of relations ensures that the correct components of each replica will be identified and aligned during synchronization. The schema that we use to synchronize this relation, `*[Phone[VAL]]`, which denotes bushes of keyed trees, each pointing to a single `Phone` edge that points to a value, implies that the result of synchronization is a proper encoding of an entry in the relation. In particular, it ensures that no entry contains two values under the `Phone` edge.

For this example, synchronizing o, a, and b successfully removes `Chris`, adds `Jo`, and leaves `Pat` with a deeper schema domain conflict, because adding both `111-2222` and `123-4567` produces a tree that is not a value in `![{}]`.

XML

Building on either the cons cell or keyed encoding for lists, it is easy to find an encoding for XML data. The XML element

```
<tag attr1="val1" ... attrm="valm">
   subelt1 ... subeltn
</tag>
```

is encoded into a tree of this form:

$$\left\{\texttt{tag} \mapsto \left\{\begin{array}{ll}\texttt{attr1} \mapsto \texttt{val1} & \\ \quad\ldots & \\ \texttt{attrm} \mapsto \texttt{valm} & \\ \texttt{*subelts} \mapsto \left[\begin{array}{l}\texttt{subelt1} \\ \quad\ldots \\ \texttt{subeltn}\end{array}\right] & \end{array}\right\}\right\}$$

The sub-elements `subelt1` to `subeltn` are placed in a *list* under a distinguished child named `*subelts`, preserving their ordering. Attributes are encoded as unordered children, reflecting their treatment in XML. A leaf of an XML document—a "parsed character data" element containing a text string `str`—is converted to a tree of the form `{PCDATA -> str}`.

The reader may wonder why, since our goal is to handle XML data, we did not use a form of trees closer to XML's data model in the other sections of the paper, avoiding the need for such complex encodings. Indeed, an early version of Harmony took *ordered* trees as primitive, allowing us to bypass this encoding. However, we discovered that this extra structure greatly increased the complexity of formalizing and implementing our lens programming language, FOCAL. On balance, the overall complexity of the system was minimized by making the core data structure as simple as possible and doing some extra programming *in* FOCAL to deal with XML structures in encoded form.

B An Alternate Treatment of Conflicts

An earlier version of Harmony used *atomicity annotations* in the trees being synchronized to recognize conflicts and ensure well-formedness of the output replicas. The schema-based approach described in the body of the paper now seems better to us. Nevertheless, the other approach had some interesting features and it seems worth recording it in case some of the ideas may be useful later.

This material has been collected by cut-and-paste, so some of the text here repeats text from the body of the paper.

B.1 Atomicity conflicts

The data structure on which Harmony primitively operates—unordered, edge-labeled trees—lends itself to a very straightforward recursive-tree-walking synchronization algorithm. For each node, we look at the set of child labels on each side; the ones that exist only on one side have been created or deleted (depending on the original replica), and are treated appropriately, taking into account delete/modify conflicts; for the ones that exist on both sides, we synchronize recursively (this algorithm is described in more detail in Section 5). However, this procedure, as we have just described it, is too permissive: in some situations, it gives us too *few* conflicts! Consider the following example. (We revert to the fully explicit notation for trees here, to remind the reader that each "leaf value" is really just a label leading to an empty subtree.)

$$\texttt{O} \quad = \quad \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{333-4444} \mapsto \{\}\}\}\}$$

$$\texttt{A} \quad = \quad \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{111-2222} \mapsto \{\}\}\}\}$$

$$\texttt{B} \quad = \quad \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{987-6543} \mapsto \{\}\}\}\}$$

If we apply the naive synchronization algorithm sketched above to these replicas, we get:

$$\texttt{A}' \quad = \quad \texttt{B}' \quad = \quad \left\{\texttt{Pat} \mapsto \left\{\texttt{Phone} \mapsto \left\{\begin{array}{l}\texttt{111-2222} \mapsto \{\} \\ \texttt{987-6543} \mapsto \{\}\end{array}\right\}\right\}\right\}$$

The subtree labeled `333-4444` has been deleted in both replicas, and remains so in both `A'` and `B'`. The subtree labeled `111-2222` has been created in `A`, so we can propagate the creation to `B'` (there is no question of a create/create conflict here: this

edge was created just in A); similarly, we can propagate the creation of 987-6543 to A′. But this is wrong: as far as the user is concerned, Pat's phone number was *changed* in different ways in the two replicas: what's wanted is a conflict. Indeed, if the phonebook schema only allows a single number per person, then the new replica is not only not what is wanted—it is not even well formed!

We have experimented with many possible mechanisms for preventing this kind of mangling. The one described below is the one we've found to work best in terms of handling all the examples we've needed it for, with a single, fairly intuitive, mechanism.[6] Section 8 sketches an idea for a related but more powerful mechanism based on types.

We introduce a special name @, and stipulate that, during synchronization, trees reached by edges labeled @ must be completely identical; otherwise a conflict is signalled *at the parent* of @, and synchronization stops. (This is stated more precisely in Section 5.) If an entire subtree must be modified atomically, we simply insert @ as its parent, as shown in the example below. If some other structure on t must be maintained, we insert @ as a sibling of t, and encode the structure of t that must be preserved as a subtree of @, and depend upon the local lenses to maintain the necessary relationship between t and @.

If we insert @ edges above the phone numbers in all three replicas in the example,

$$\texttt{O} \quad = \quad \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{@ \mapsto \{\texttt{333-4444} \mapsto \{\}\}\}\}\}$$

$$\texttt{A} \quad = \quad \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{@ \mapsto \{\texttt{111-2222} \mapsto \{\}\}\}\}\}$$

$$\texttt{B} \quad = \quad \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{@ \mapsto \{\texttt{987-6543} \mapsto \{\}\}\}\}\}$$

then the rule for @ yields a conflict and the sychronizer returns the original replicas unchanged.

B.2 Synchronization

A key feature of Harmony's design is that it offers just *one* algorithm for actually performing synchronization; the behavior of the synchronization tool as a whole is tuned for particular applications not by changing the functioning of this algorithm, but by writing lenses that format concrete application data as abstract trees of a suitable shape. In particular, lenses can control how the synchronizer behaves by (1) "pre-aligning" information so that, for example, key fields are moved high in the abstract tree, where they determine the "path" by which records are reached by the synchronization algorithm, and (2) choosing where to insert @ labels to control the atomicity of synchronization.

After introducing some notation for trees, we describe the core algorithm and relate its behavior to a formal specification.[7] We close the section by establishing some key invariants for synchronization of structures involving atomicity.

B.2.1 Notation

We write $\mathcal{N}$ for the set of character strings and $\mathcal{T}$ for the set of unordered, edge-labeled trees whose labels are drawn from $\mathcal{N}$ and where the labels of the immediate children of each node are pairwise distinct.

A tree can be viewed as a partial function from names to other trees; we write $t(n)$ for the immediate subtree of t labeled with the name n. We write $dom(t)$ for the domain of a tree t—i.e. the set of the names of its immediate children.

[6] A review of our earlier attempts may be of interest to some readers. We started by labeling trees with an *atomic bit*. If a subtree were atomic then we raised a conflict unless updates occured on only one replica. This definition was too strict. All we needed to preserve was the structure (the well-formedness) of each replica, but this definition did not allow non-conflicting updates to values in the subtree. At the time, the only structural property we used in practice was limiting certain trees to a single child; therefore we labeled trees as SINGLETON to enforce that restriction. Synchronization that resulted in multiple children for such a tree would, instead, raise a conflict. Eventually, we needed richer encodings and correspondingly more general notions of atomicity conflicts. Our next attempt was to tag a tree, A, *atomic* by giving it a child @, but only raise conflicts if the domains (the labels of the immediate children of A) differed between replicas. This was unsatisfying, because it sometimes discovered conflicts "too late". For example, we wanted our list encoding to trigger a conflict at the root of the "cons cell" when the head was modified incompatibly on both archives. However, the conflict was raised at the head, letting the synchronizer inspect the tail and occasionally generate ill-formed lists. Our solution was to push the conflict one level up the tree (a form of one-deep lookahead). If the domain of $A(@)$ did not equal the domain of $B(@)$, then we triggered the conflict at the *parents* of @, namely at A and B. Once again, this resulted in discovering conflicts "too late" when we looked at richer encodings — if the schema conflict occured two levels deep, we still wanted to trigger the conflict at the root of the atomic structure. Our current definition cleanly separates the schema definition (an arbitrarily deep representation under the @ child) from the data (the other children of the root). We rely on the lenses to locally maintain the consistency between the schema and the data.

[7] This section differs from previously circulated manuscripts of this paper in two significant respects: we show explicitly how the result archive O is calculated by the algorithm, and we have changed the details of the treatment of the @ label to obtain a correct handling of the result archive in the case of conflicts involving ordered data.

```
sync(O, A, B) =
  if A = B then (A,A,B)                                  -- equal replicas: done
  else if A = O then (B,B,B)                             -- no change to A: propagate B
  else if B = O then (A,A,A)                             -- no change to B: propagate A
  else if O = X then (O,A,B)                             -- unresolved conflict
  else if A = missing then (X,A,B)                       -- delete/modify conflict
  else if B = missing then (X,A,B)                       -- delete/modify conflict
  else if @ in dom(A) and @ not in dom(B)
       or @ in dom(B) and @ not in dom(A)
       or @ in dom(A) and @ in dom(B) and A(@) != B(@)
       then (X,A,B)                                      -- atomicity conflict
  else                                                   -- else proceed recursively
    (O',A',B')
      where O'(k),A'(k),B'(k) = sync(O(k),A(k),B(k))
      for all k in dom(A) union dom(B)
```

Figure 4: Core Synchronization Algorithm

When $n \notin dom(t)$, we define $t(n)$ to be the "missing tree" $\perp$. A *replica* may be either a tree or $\perp$. Our synchronization algorithm below takes replicas as inputs and returns replicas as outputs; regarding "missing" as a possible replica state allows the algorithm to treat creation and deletion uniformly. By convention, we take $dom(\perp) = \emptyset$.

The archive that is stored between synchronizations must keep track of where conflicts have occurred. To this end, we introduce a special "pseudo-tree" $\mathcal{X}$ representing a conflict. We write $\mathcal{T}_{\mathcal{X}}$ for the set of extended trees that may contain $\mathcal{X}$ as a subtree. We write $\mathcal{T}_{\perp}$ for the set $\mathcal{T} \cup \{\perp\}$ and $\mathcal{T}_{\mathcal{X}\perp}$ for the set $\mathcal{T}_{\mathcal{X}} \cup \{\perp\}$; we call the latter set *archives*. We define $dom(\mathcal{X}) = \{n_{\mathcal{X}}\}$, where $n_{\mathcal{X}}$ is a special name that cannot occur in ordinary trees.

A *path* is a sequence of names. We write $\bullet$ for the empty path and p/q for the concatenation of paths p and q. The *contents* of a tree, replica, or archive t at a path p, written $t(p)$, is defined as follows:

$$\begin{aligned} t(\bullet) &= t \\ t(p) &= \mathcal{X} && \text{if } t = \mathcal{X} \\ t(n/p) &= (t(n))(p) && \text{if } t \neq \mathcal{X} \text{ and } n \in dom(t) \\ t(n/p) &= \perp && \text{if } t \neq \mathcal{X} \text{ and } n \notin dom(t) \end{aligned}$$

In the proofs we often proceed by induction on the height of a tree. We define $height(\perp) = height(\mathcal{X}) = 0$ and the height of any other tree to be $height(t) = 1 + \texttt{max}(\{height(t(k)) \mid k \in dom(t)\})$. Note that the height of the empty tree (a node with no children) is 1, to avoid confusing it with the missing or the conflict tree.

B.2.2 Synchronization Algorithm

We now describe our synchronization algorithm, depicted in Figure 2. Its general structure is the following: we first check for trivial cases (replicas being equal to each other or unmodified), then we check for conflicts, and in the general case we recurse on each child label and combine the results.

In practice, synchronization will be performed repeatedly, with additional updates applied to one or both of the replicas between synchronizations. To support this, a new archive needs to be constructed by the synchronizer. Its calculation is straightforward: we use the synchronized version at every path where the replicas agree and insert a conflict marker $\mathcal{X}$ at paths where replicas are in conflict.

Formally, the algorithm takes as inputs an archive O and two current replicas A and B and outputs a new archive O' and two new replicas A' and B'. Any of the inputs and outputs may be $\perp$, which stands for a completely missing (or deleted) replica, and both the input and output archive may contain the special conflict tree $\mathcal{X}$—that is, the type of `sync` is $\mathcal{T}_{\mathcal{X}\perp} \times \mathcal{T}_{\perp} \times \mathcal{T}_{\perp} \to \mathcal{T}_{\mathcal{X}\perp} \times \mathcal{T}_{\perp} \times \mathcal{T}_{\perp}$.

In the case where A and B already agree (they are both the same tree or both $\perp$), they are immediately returned, and the new archive is set to their value. If one of the replicas is unchanged (equal to the archive), then all the changes in the other replica can safely be propagated, so we simply return three copies of it as the result replicas and archive. Otherwise, both replicas have changed, in different ways. In this case, if the archive is a conflict, then the conflict is preserved and A and B are returned unmodified. If

one replica is missing (it has been deleted), then we have a *delete/modify conflict* since the other replica has changed, so we simply return the original archive and replicas.

If both A and B are atomic (i.e., both have a child named @), we check whether their subtrees rooted at @ are identical. If not, then an *atomicity conflict* is generated and we return the inputs unchanged. If only one of A and B is marked atomic, then an *atomicity conflict* is again signalled (this should never happen if the lenses are written correctly).

Finally, in the general case, the algorithm recurses. In this case, subtrees under identical names are synchronized together.

B.2.3 Safety and Maximality

We now give a formal specification of the properties we want our synchronization algorithm to satisfy and prove that it does indeed satisfy them. We follow the basic approach used for specifying the Unison file synchronizer [29], adapting it to our setting and extending it to describe the generation of the new archive.

Our specification is based on a notion of *local equivalence*, that relates two trees (or replicas or archives) if their top-level nodes are similar— i.e., roughly, if both are present or both are missing.

B.1 Definition [Local equivalence]: We say that two elements of $\mathcal{T}_{\mathcal{X}\perp}$ are locally equivalent, written $t \sim t'$, iff

- $t = t' = \mathcal{X}$; or
- $t = t' = \perp$; or
- t and t' are proper trees with $@ \notin dom(t) \cup dom(t')$; or
- t and t' are proper trees with $@ \in dom(t) \cap dom(t')$ and $t(@) = t'(@)$.

A first approximation of local equivalence is the presence of information: two trees are locally equivalent iff both are conflicting, both are missing, or neither is missing. Using this notion of local equivalence, one can prove that two trees are identical iff they are locally equivalent at all paths. However this definition is too local to capture the notion of atomicity, which considers not just a node, but a larger structure (the whole subtree below @). Thus our definition of local equivalence requires the less local constraint either that neither tree be atomic or else that both trees be atomic and both @ children identical. This results in more conflicts in the case of atomic trees.

B.2 Lemma: The local equivalence relation is an equivalence.

Proof: The definition is obviously reflexive and symmetric. For transitivity, choose any $t, t', t'' \in \mathcal{T}_{\mathcal{X}\perp}$ such that $t \sim t'$ and $t' \sim t''$. We show $t \sim t''$ by cases on the local equivalence rule applied to derive $t \sim t'$.

- If $t = t' = \mathcal{X}$, then as $t' \sim t''$ we must have $t'' = \mathcal{X}$, hence $t \sim t''$.
- If $t = t' = \perp$, then as $t' \sim t''$ we must have $t'' = \perp$, hence $t \sim t''$.
- If both t and t' are proper trees and neither is atomic, then by $t' \sim t''$, we know that t'' is not $\perp$, is not $\mathcal{X}$, and cannot be atomic (as t' is not atomic). Hence we have $t \sim t''$.
- If both t and t' are atomic and $t(@) = t'(@)$, then by $t' \sim t''$, we must have t'' atomic and $t'(@) = t''(@)$. Hence we have $t \sim t''$. □

In the following we silently rely on the fact that $\sim$ is an equivalence relation.

B.3 Definition [Conflict]: We say that o, a, and b conflict, written $\mathsf{conflict}(o, a, b)$, if

$$((o = \mathcal{X}) \wedge (a \neq b)) \vee ((a \nsim b) \wedge (o \neq a) \wedge (o \neq b))$$

Intuitively, a and b conflict if there is a conflict recorded in the archive that has not been resolved, or if they are not locally equivalent and both have changed since the state recorded in the archive. The conflicts described in Section 4, such as atomicity or delete/delete conflicts, are captured by the definition of local equivalence.

A *run* of a synchronizer is a six-tuple (o, a, b, o', a', b') of trees, representing the original synchronized state (o), the states of the two replicas before synchronization (a, b), the new archive (o'), and the states of the replicas after synchronization (a', b').

We now state the properties our synchronizer must satisfy: the result of synchronization must reflect all user changes, it must not include changes that do not come from either replica, and trees under a conflicting node should remain untouched.

B.4 Definition [Local safety]: A run is *locally safe* iff

1. It never overwrites changes locally:

$$o \nsim a \implies a' \sim a$$
$$o \nsim b \implies b' \sim b$$

2. It never "makes up" content locally:

$$a \nsim a' \implies b \sim a'$$
$$b \nsim b' \implies a \sim b'$$
$$o' \neq \mathcal{X} \implies o' \sim a' \wedge o' \sim b'$$

3. It stops at conflicting paths (leaving replicas in their current states and recording the conflict):

$$\mathsf{conflict}(o, a, b) \implies (a' = a) \wedge (b' = b) \wedge (o' = \mathcal{X})$$

B.5 Definition [Safe run]: A run (o, a, b, o', a', b') is *safe*, written $\mathit{safe}(o, a, b, o', a', b')$, iff for every path p, the sub-run $(o(p), a(p), b(p), o'(p), a'(p), b'(p))$ is locally safe.

B.6 Lemma: The identity run $(o, a, b, \mathcal{X}, a, b)$ is safe.

Proof: Let p be a path. We have $\mathcal{X}(p) = \mathcal{X}$. As $a' = a$, $b' = b$, and $o' = \mathcal{X}$, local safety conditions (1,2) are satisfied at every path. As $a' = a$, $b' = b$, and $o' = \mathcal{X}$, local safety condition (3) is also satisfied at every path. □

B.7 Lemma: Let (o, a, b, o', a', b') be a safe run. For any path p, the run $(o(p), a(p), b(p), o'(p), a'(p), b'(p))$ is safe.

Proof: Immediate by definition of safety. □

Of course, safety is not all we want. We also want to insist that a good synchronizer should propagate as many changes as possible.

B.8 Definition [Maximality]: A safe run (o, a, b, o', a', b') is *maximal* iff it propagates at least as many changes as any other safe run, i.e.

$$\forall o'', a'', b''.\ \mathit{safe}(o, a, b, o'', a'', b'') \implies \begin{cases} \forall p.\ a''(p) \sim b''(p) \implies a'(p) \sim b'(p) \\ \forall p.\ o''(p) \neq \mathcal{X} \implies o'(p) \neq \mathcal{X}. \end{cases}$$

We can now state precisely what we mean by claiming that Harmony's synchronization algorithm is correct.

B.9 Theorem: If $\mathtt{sync}(o, a, b)$ evaluates to (o', a', b'), then (o, a, b, o', a', b') is maximal.

Proof: We proceed by induction on the sum of the depth of o, a, and b, with a case analysis according to the first rule in the algorithm that applies.

case $a = b$**:** We need to show that (o, a, a, a, a, a) is maximal. We first check that it is safe. Let p be a path. Local safety condition (1) is satisfied since $a'(p) = a(p) \sim a(p)$ and $b'(p) = a(p) = b(p) \sim b(p)$. Local safety condition (2) is satisfy for the same reasons, and because $o'(p) = a(p) \sim a(p) = a'(p) = b'(p)$. As we have $a = b$, we have $a(p) \sim b(p)$ hence there is no conflict at path p.

The first condition for maximality is immediate as for all paths p, $a'(p) = a(p) \sim a(p) = b'(p)$. The second condition is also satisfied, since $o' = a$, hence we have $o'(p) \neq \mathcal{X}$ for all paths p.

case $a = o$**:** We need to show that (o, o, b, b, b, b) is maximal. We first check that it is safe. Let p be a path. Local safety condition (1) is satisfied since $a = o$ and $b' = b$. Local safety condition (2) is satisfied since $a' = b$, since $b = b'$, and since $o' = b = a' = b'$. Finally, $o(p)$, $a(p)$, and $b(p)$ cannot conflict since $a = o$ and $o' = b \neq \mathcal{X}$ at all paths.

The first condition for maximality is immediate, since $a'(p) \sim b'(p)$ for all paths p. The second condition is also satisfied, since $o' = b$, hence we have $o'(p) \neq \mathcal{X}$ for all paths p.

case $b = o$**:** Identical to the previous case, inverting the roles of a and b.

case $o = \mathcal{X}$: By Lemma B.6, the run $(\mathcal{X}, a, b, \mathcal{X}, a, b)$ is safe. We now show that we have $\mathsf{conflict}(\mathcal{X}, a, b)$. This is immediately the case since we know that $a \neq b$ (as the first case of the algorithm did not apply). By safety condition 3, the only safe run is $(\mathcal{X}, a, b, \mathcal{X}, a, b)$, hence it is maximal.

case $a = \bot$: By lemma B.6, the run is safe. We now prove that it is maximal. To this end, we first prove that we have $\mathsf{conflict}(o, a, b)$. As no previous rule applies, we must have $b \neq a = \bot$, $o \neq a = \bot$, and $b \neq o$. Since $a = \bot$ and $b \neq \bot$, we also have $a \nsim b$. Hence we have $\mathsf{conflict}(o, a, b)$.

As before, by safety condition 3, the only safe run is $(\mathcal{X}, a, b, \mathcal{X}, a, b)$, hence it is maximal.

case $b = \bot$: Identical to the previous case, inverting the roles of a and b.

atomicity conflict case: This run being the identity run, it is safe by lemma B.6.

To prove maximality, we proceed as in the previous cases, proving that we have $\mathsf{conflict}(o, a, b)$.

Since previous cases of the algorithm are not satistfied, we immediately have $o \neq a$ and $o \neq b$. We now prove that $a \nsim b$.

First of all, we have $a \neq \mathcal{X}$ and $b \neq \mathcal{X}$.

As the previous cases of the algorithm are not satisfied, we have $a \neq \bot$ and $b \neq \bot$, discarding the second case of the definition of local equivalence. As we have $@ \in dom(a)$ or $@ \in dom(b)$ (or both), the third case of the definition of $\sim$ cannot apply. Finally, in the case where $@ \in dom(a) \cap dom(b)$, as we have $a(@) \neq b(@)$, the fourth case of the definition cannot apply. Thus we have $a \nsim b$.

We conclude by local safety condition (3) that the only safe run is the identity run.

recursive case: The induction hypothesis immediately tells us that this run is locally safe at every path except possibly the root. We now check that it is also locally safe at the root.

We first show that $a \sim b$. Since previous cases of the algorithm do not apply, we have $a \neq \bot$ and $b \neq \bot$. If neither a nor b is atomic, we have $a \sim b$. If one is atomic, then, as the atomicity conflict case of the algorithm did not apply, so is the other and we have $a(@) = b(@)$, thus $a \sim b$.

We now show $a \sim a'$, $a' \sim b'$, and $o' \sim a'$. As a', o', and b' are built as the result of the recursive calls, we have $a' \neq \bot$, $b' \neq \bot$, $o' \neq \bot$, and $o' \neq \mathcal{X}$ (recall the difference between the empty tree and the missing tree). So these equivalences depend on the atomicity of the input and output of the algorithm. We now consider the atomicity of a and b, study the result of the recursive call of the algorithm on the $atomic$ child. We describe each case as the tuple (t_a, t_b) meaning $a(@) = t_a$ and $b(@) = t_b$.

$(\bot, \bot)$**:** This case is immediate, as the synchronization under $@$ yields $(\bot, \bot, \bot)$, hence neither a, a', b', nor o' is atomic.

$(t_a, \bot)$ and $(\bot, t_b)$**:** This case cannot occur as it is an atomicity conflict.

(t_a, t_b)**:** Since there was no atomicity conflict, we have $t_a = t_b$. Hence synchronization succeeds for the $@$ child (using the first branch of the algorithm) and we have $a'(@) = a(@) = b(@) = b'(@) = o'(@) = t_a$Hence $a \sim a' \sim b' \sim o'$.

Similarly, we show that $b \sim b'$ and that $o' \sim b'$.

As $a \sim a'$, $b \sim b'$, $o' \sim a'$, and $o' \sim b'$, local safety conditions (1,2) are immediately satisfied. Since $a \sim b$, and since $o \neq \mathcal{X}$ (otherwise the recursive case of the algorithm would not be reached), there is no conflict at the root and local safety condition (3) is also satisfied. We conclude that the run is safe.

To conclude, we must also prove that this run is maximal. So let (o, a, b, o'', a'', b'') be another safe run. Let p be a path.

- If p is not the empty path, then it may be decomposed as k/p'. By induction, the run $(o(k), a(k), b(k), o'(k), a'(k), b'(k))$ is maximal. By Lemma B.7, $(o(k), a(k), b(k), o''(k), a''(k), b''(k))$ is a safe run. We have $a''(p) = a''(k/p') = (a''(k))(p')$, and $b''(p) = b''(k/p') = (b''(k))(p')$.
  - If $a''(p) \sim b''(p)$, then $a''(p) = (a''(k))(p') \sim (b''(k))(p') = b''(p)$, hence we have (by maximality of the run $(o(k), a(k), b(k), o'(k), a'(k), b'(k))$) that $(a'(k))(p') \sim (b'(k))(p')$, hence $a'(p) \sim b'(p)$.
  - If $o''(p) \neq \mathcal{X}$, then $(o''(k))(p') \neq \mathcal{X}$, hence we have (by maximality of the run $(o(k), a(k), b(k), o'(k), a'(k), b'(k))$) that $(o'(k))(p') \neq \mathcal{X}$, hence $o'(p) \neq \mathcal{X}$.
- Let us now assume that p is the empty path.
  - We have $a' \sim b'$, so the first maximality condition is satisfied at the root.
  - We have $o' \neq \mathcal{X}$, so the second maximality condition is satisfied at the root. □

B.2.4 Properties of Atomicity

We next collect some properties that are guaranteed by atomicity and used in our encodings. To this end, we first need a few additional definitions and lemmas about the synchronization algorithm.

B.10 Lemma: Let o be any archive, let a and b be two replicas different from $\bot$, and let $(o', a', b') = \texttt{sync}(o, a, b)$. Then o', a', and b' are all different from $\bot$.

Proof: We proceed by cases on the clause in the algorithm that applies.

case $a = b$**:** In this case $o' = a' = a \neq \bot$ and $o' = b' = b \neq \bot$.

case $a = o$**:** In this case $o' = a' = b' = b \neq \bot$.

case $b = o$**:** In this case $o' = a' = b' = a \neq \bot$.

case $o = \mathcal{X}$**:** In this case $o' = \mathcal{X} \neq \bot$, $a' = a \neq \bot$, and $b' = b \neq \bot$.

case $a = \bot$**:** Can't happen (we assumed $a \neq \bot$).

case $b = \bot$**:** Can't happen.

atomicity conflict case: In this case $a' = a \neq \bot$ and $b' = b \neq \bot$, and $o' = \mathcal{X} \neq \bot$.

recursive case: As o', a', and b' are trees explicitly built in this case, they cannot be $\bot$. □

B.11 Definition [Tree Prefix]: The relation $<$ on $\mathcal{T} \times \mathcal{T}$ is defined as the smallest relation such that:

- $\{\} < t$ for any $t \in \mathcal{T}$;
- if $dom(t_1) = dom(t_2)$ and $\forall k \in dom(t_1).t_1(k) < t_2(k)$, then $t_1 < t_2$.

We write $t \setminus n$ for the tree $\{k \mapsto t(k) \mid k \in dom(t) \setminus \{n\}\}$ whose n child and accompanying subtree have been removed.

B.12 Lemma: Suppose $t \in \mathcal{T}$ and $(o', a', b') = \texttt{sync}(o, a, b)$. If $t < a$ and $t < b$, then $t < a'$ and $t < b'$.

Proof: By induction on the size of t. First, we remark that since $t < a$ and $t < b$, we have $a \neq \bot$ and $b \neq \bot$, hence $a' \neq \bot$ and $b' \neq \bot$ by Lemma B.10.

The base case $t = \{\}$ is immediate as neither a' nor b' is missing.

For the inductive case, we proceed by cases on the branch of the algorithm used, using the induction hypothesis only for the recursive branch.

case $a = b$**:** Immediate since $a' = b' = a = b$.

case $a = o$**:** Immediate since $a' = b' = b$.

case $b = o$**:** Immediate since $a' = b' = a$.

conflict cases: Immediate since $a' = a$ and $b' = b$.

recursive case: Let $k \in dom(t)$. Then we have $t(k) < a(k)$ and $t(k) < b(k)$. Hence by induction we have $t(k) < a'(k)$ and $t(k) < b'(k)$. Moreover neither $a'(k)$ nor $b'(k)$ is missing, hence $k \in dom(a')$ and $k \in dom(b')$.

Let $k \notin dom(t)$, then $k \notin dom(a')$ and $k \notin dom(b')$. By the first branch of the algorithm when synchronizing under k, we have $k \notin dom(a')$ and $k \notin dom(b')$. Thus we conclude that $dom(a') = dom(b') = dom(t)$, and that $t < a'$ and $t < b'$. □

B.13 Lemma: Suppose a and b are atomic trees such that @ does not occur in $dom(a(@))$ or $dom(b(@))$, and let $(o', a', b') = \texttt{sync}(o, a, b)$. If $a(@) < a \setminus @$ and $b(@) < b \setminus @$, then we have $a'(@) < a' \setminus @$ and either $a'(@) = a(@)$ or $a'(@) = b(@)$.

Proof: We proceed by cases on the branch taken by the algorithm.

case $a = b$**:** Immediate since $a' = a$.

case $a = o$**:** Immediate since $a' = b$.

case $b = o$**:** Immediate since $a' = a$.

conflict cases: Immediate since $a' = a$.

recursive case: In this case we know that $a(@) = b(@) = t$, hence by synchronizing under the @ child we have $a'(@) = t$. If $a(@) = \{\}$, then the result is immediate, as a' is not $\bot$. Otherwise, we have $dom(a(@)) = dom(a \backslash @) = dom(b \backslash @) = D$. Let $k \in D$, then we have $t(k) < a(k)$ and $t(k) < b(k)$. By Lemma B.12, we have $t(k) < a'(k)$. This also implies that $a'(k) \neq \bot$, hence $k \in dom(a')$. Hence we have $dom(a(@)) \subseteq dom(a' \setminus @)$. Let k be a name that is neither @ nor in $dom(a(@))$, then $k \notin dom(a)$ and $k \notin dom(b)$, hence (by the first case of the synchronization algorithm with $a(k) = b(k) = \bot$), we have $k \notin dom(a')$. Hence we have $dom(a(@)) = dom(a' \setminus @)$ and for all $k \in dom(a(@))$, $(a(@))(k) < (a' \setminus @)(k)$, thus $a'(@) = a(@) < a' \setminus @$. □

B.14 Lemma: Suppose a and b are atomic and $(o', a', b') = \texttt{sync}(o, a, b)$. If $a(@) \neq b(@)$, then either $a' = a$ or $a' = b$.

Proof: We proceed by cases on the branch taken by the algorithm.

case $a = b$**:** This case cannot arise, since $a(@) \neq b(@)$.

case $a = o$**:** Immediate since $a' = b$.

case $b = o$**:** Immediate since $a' = a$.

conflict cases: Immediate since $a' = a$.

recursive case: This case cannot occur as there is an atomicity conflict. □

Lemmas B.13 and B.14 give us a useful framework for proving that our synchronization algorithm preserves particular atomic encodings of data structures. That is, assuming that we can make a purely *local* guarantee that any modification affecting well-formedness of an encoding is reflected by a change to the subtree under @, then synchronization of two instances of an encoded data structure is guaranteed to produce a valid encoded data structure. This follows because the lemmas, as a pair, prove that the only way the synchronization algorithm can reach the recursive branch (the only one in which synchronization can merge pieces of a and b) is if $a(@) = b(@)$.

B.3 Atomicity in Action

We now discuss the treatment of ordered structures such as lists using atomicity annotations.

Extensible Tuples

An extensible tuple is an application data structure consisting of an arbitrary-length sequence of elements, on which the possible "edits" may be thought of as consisting of adding and deleting elements at the end and/or changing the internals of individual elements. We will write

$$\begin{pmatrix} \mathtt{t}_1 \\ \mathtt{t}_2 \\ \vdots \\ \mathtt{t}_n \end{pmatrix}$$

—or, in linear form, $(t_1 \ldots t_n)$— for the tree representing the extensible tuple with elements t_1 through t_n.

One might initially hope that the encoding of ordinary (fixed-width) tuples from the previous section would also work for extensible tuples. However, in the presence of conflicts, our synchronization algorithm may produce outputs that do not conform to the abstract schema. For example, if the inputs to the synchronizer are

$$O = \left\{ \begin{matrix} \mathtt{1} \mapsto \mathtt{x} \\ \mathtt{2} \mapsto \mathtt{y} \\ \mathtt{3} \mapsto \mathtt{z} \\ \mathtt{4} \mapsto \mathtt{w} \end{matrix} \right\}$$

$$A = \{\}$$

$$B = \left\{ \begin{matrix} \mathtt{1} \mapsto \mathtt{x} \\ \mathtt{2} \mapsto \mathtt{y} \\ \mathtt{3} \mapsto \mathtt{c} \\ \mathtt{4} \mapsto \mathtt{w} \end{matrix} \right\}$$

(i.e., all the elements have been deleted from A, while the third has been changed in B), then the output will be:

$$O' = \{\texttt{3} \mapsto \mathcal{X}\}$$
$$A' = \{\}$$
$$B' = \{\texttt{3} \mapsto \texttt{c}\}$$

The synchronization algorithm does not understand (because our abstract schema does not specify) that a tuple with a third element but no first or second makes no sense.

A better idea is to represent extensible tuples using nested pairs, as in the standard "cons cell" representation of lists from Lisp. Roughly, the representation we want is this:

$$\left\{\begin{array}{l}\texttt{head} \mapsto \texttt{t}_1 \\ \texttt{tail} \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{t}_2 \\ \texttt{tail} \mapsto \left\{\ldots \mapsto \left\{\begin{array}{l}\texttt{head} \mapsto \texttt{t}_n \\ \texttt{tail}\end{array}\right\}\right\}\end{array}\right\}\end{array}\right\}$$

That is, a tree t represents an extensible tuple iff it is empty or has exactly two children, one named `head` and another named `tail`, with $t(\texttt{tail})$ also an extensible tuple. However, we again need to make sure that the structure of the encoding is preserved by synchronization. Consider, for instance, the following inputs:

$$O = \left\{\begin{array}{l}\texttt{head} \mapsto \{\texttt{Pat} \mapsto \texttt{333-4444}\} \\ \texttt{tail} \mapsto \{\}\end{array}\right\}$$
$$A = \{\}$$
$$B = \left\{\begin{array}{l}\texttt{head} \mapsto \{\texttt{Pat} \mapsto \texttt{111-2222}\} \\ \texttt{tail} \mapsto \{\}\end{array}\right\}$$

The changes to the first element result in a delete/modify conflict, but the deletion of the remainder successfully synchronizes, yielding a malformed structure as the new version of replica B (A' is equal to A):

$$B' = \{\texttt{head} \mapsto \{\texttt{Pat} \mapsto \texttt{111-2222}\}\}$$

In order to avoid this problem, the abstract schema for extensible tuples needs to encode the constraint that the domain of a tree representing a tuple must be treated atomically. To each node, we add an @ child describing the local tuple structure—$\{@ \mapsto \{\}\}$ for the end of the tuple (i.e., nil) and $\{@ \mapsto \{\texttt{head}, \texttt{tail}\}\}$ for internal nodes (cons cells). The synchronization now results in an atomic conflict at the root of the tree.

$$O = \left\{\begin{array}{l}@ \mapsto \left\{\begin{array}{l}\texttt{head} \\ \texttt{tail}\end{array}\right\} \\ \texttt{head} \mapsto \{\texttt{Pat} \mapsto \texttt{333-4444}\} \\ \texttt{tail} \mapsto \{@ \mapsto \{\}\}\end{array}\right\}$$
$$A = \{@ \mapsto \{\}\}$$
$$B = \left\{\begin{array}{l}@ \mapsto \left\{\begin{array}{l}\texttt{head} \\ \texttt{tail}\end{array}\right\} \\ \texttt{head} \mapsto \{\texttt{Pat} \mapsto \texttt{111-2222}\} \\ \texttt{tail} \mapsto \{@ \mapsto \{\}\}\end{array}\right\}$$

In non-conflicting situations, this abstract schema produces the intuitively expected propagation of updates. Consider for instance the following pre-synchronization state: $O = (a; b; c)$, $A = (a; b'; c)$, and $B = (a; b; c'; d)$. Synchronization returns to the expected state $[a; b'; c'; d]$. We now show that the set of encodings of extensible tuples is closed under synchronization.

B.15 Proposition: Let $o \in \mathcal{T}_{\mathcal{X}\perp}$, let a and b be well-formed encodings of extensible tuples as trees, and let $(o', a', b') = \texttt{sync}(o, a, b)$. Then a' and b' are also well-formed encodings of extensible tuples.

Proof: A tree representing an extensible tuple is well formed if either it is empty or (i) it has 3 children @, `*h`, and `*t`, (ii) the children of @ are `*h` and `*t`, and (iii) the subtree under `*t` is itself a well-formed extensible tuple. Without loss of generality, suppose a is not longer than b. We proceed by induction on the length of a. For the base case, let a be the empty tuple (i.e., the empty tree). If b is also empty, then $a' = b' = a$ and we are done. If b is non-empty, it follows that $a(@) \neq b(@)$. By Lemma B.14, either $a' = a$ or $a' = b$ (and, equivalently for b'), and again the proposition holds. For the induction case, suppose the proposition

holds for all a with length less than n. By Lemma B.13 $a'(@) = a(@) = b(@)$, guaranteeing that $a'(@)$ is in valid form. Moreover, by the same Lemma, $a'(@) < a' \setminus @$ so that a' contains both `*h` and `*t`. Moreover, a' cannot contain any other child k, unless $k \in dom(a)$ or $k \in dom(b)$. It remains only to show that $a'(\texttt{*t})$ is a well-formed extensible tuple. But $a'(\texttt{*t})$ is the result of evaluating $\texttt{sync}(o(\texttt{*t}), a(\texttt{*t}), b(\texttt{*t}))$, which is well formed by the induction hypothesis. □

Lists

Ordered data in many applications relies on *relative position*. Detecting changes in relative position is a global process and our synchronization algorithm is essentially local, so our algorithm in its current form is not well-suited to this form of synchronization. The best we can hope for is to behave safely—i.e., never to produce mangled or ill-formed replicas—while propagating changes successfully just in some simple situations where it is absolutely clear what to do. (Fortunately, these simple situations are common in practice. For example, if a list has been edited only in one of the replicas—or if just the elements of the list have been edited, without changing the list structure—we can safely propagate the changes to the other replica.)

The extensible tuple schema proposed above is inadequate for real lists: it may lead to conflicting cases where the conflict is detected too late. To see why, consider the following example.

$$O = [\{\texttt{Pat} \mapsto \texttt{333}\}; \{\texttt{Chris} \mapsto \texttt{888}\}]$$
$$A = [\{\texttt{Chris} \mapsto \texttt{123}\}]$$
$$B = [\{\texttt{Pat} \mapsto \texttt{333}\}; \{\texttt{Chris} \mapsto \texttt{888}\}; \{\texttt{Jo} \mapsto \texttt{314}\}]$$

The first element of the list is successfully synchronized, but a delete/modify conflict is detected when synchronizing the rest of the list. The result of synchronization for B is:

$$B' = [\{\texttt{Chris} \mapsto \texttt{123}\}; \{\texttt{Chris} \mapsto \texttt{888}\}; \{\texttt{Jo} \mapsto \texttt{314}\}]$$

This result is probably unsatisfactory, since the list now contains two entries for `Chris`.

In order to avoid these cases, we propose an alternative schema, called *atomic list schema*, for lists whose relative order matters. This schema allows the domain of an element of the list to be different in both replicas only when the element and the rest of the list have not changed in one replica. To this end, the atomic child includes the list element itself, to guarantee that identical elements are synchronized together, as in:

$$O = \left\{ \begin{array}{l} @ \mapsto \left\{ \begin{array}{l} \texttt{head} \mapsto \texttt{Pat} \\ \texttt{tail} \end{array} \right\} \\ \texttt{head} \mapsto \{\texttt{Pat} \mapsto \texttt{333}\} \\ \texttt{tail} \mapsto \left\{ \begin{array}{l} @ \mapsto \left\{ \begin{array}{l} \texttt{head} \mapsto \texttt{Chris} \\ \texttt{tail} \end{array} \right\} \\ \texttt{head} \mapsto \{\texttt{Chris} \mapsto \texttt{888}\} \\ \texttt{tail} \mapsto \{@ \mapsto \{\}\} \end{array} \right\} \end{array} \right\}$$

Intuitively, elements of the list are identified by their domain, and synchronization proceeds until a trivial case applies (unchanged replica or identical replicas), or when the two replicas disagree on the domain of one element, resulting in an atomicity conflict. In the previous example, this would for instance be the case at the very beginning of synchronization.

We write $[t_1 \ldots t_n]$ for the tree representing the ordered list of elements t_1 through t_n.

B.16 Proposition: Let $o \in \mathcal{T}_{\mathcal{X}\perp}$, let a and b be well-formed encodings of lists as trees, and let $(o', a', b') = \texttt{sync}(o, a, b)$. Then a' and b' are also well formed encodings of lists.

Proof: A list encoding t is well formed if either t is the empty tree or else (i) t has three children @, `*h`, and `*t`, (ii) $dom(t(@)) = \{\texttt{*h}, \texttt{*t}\}$, (iii) $t(@)(*h) < t(\texttt{*h})$, and (iv) $t(\texttt{*t})$ is itself a well-formed list. Without loss of generality, suppose a is not longer than b. We proceed by induction on the length of a. For the base case, suppose a is the empty list. If b is also empty, then $a' = b' = a$, and we are done. If b is non-empty, it follows that $a(@) = \perp \neq b(@)$. By Lemma B.14, either $a' = a$ or $a' = b$ (and equivalently for b'), and again the proposition holds. For the induction case, suppose the proposition holds for all a with length less than n. By Lemma B.13, either $a'(@) = a(@)$ or $a'(@) = b(@)$, both of which are already known to be well-formed subtrees of @ under the encoding of lists. Moreover, $a'(@) < a' \setminus @$ so by the same lemma, a' contains both `*h` and `*t`, and also $a'(@)(\texttt{*h}) < a'(\texttt{*h})$. (The same is true of b'.) a' cannot contain any other child k, unless $k \in dom(a)$ or $k \in dom(b)$. It remains only to show that $a'(\texttt{*t})$ is a well-formed list. But $a'(\texttt{*t})$ is the result of evaluating $\texttt{sync}(o(\texttt{*t}), a(\texttt{*t}), b(\texttt{*t}))$, which is well formed by the induction hypothesis. □