Bringing Harmony to Optimism

An Experiment in Synchronizing Heterogeneous Tree-Structured Data

Benjamin C. Pierce, Alan Schmitt, and Michael B. Greenwald

Technical Report MS-CIS-03-42
Department of Computer and Information Science
University of Pennsylvania

March 18, 2004

Abstract

Increased reliance on optimistic data replication has led to burgeoning interest in tools and frameworks for *synchronizing* disconnected updates to replicated data. To better understand the issues underlying the design of *generic* and *heterogeneous* synchronizers, we have implemented an experimental framework, called Harmony, that can be used to build synchronizers for tree-structured data stored in a variety of concrete formats.

We present Harmony's architecture, formalize its key components (a simple core synchronization algorithm together with a set of user-defined mappings between diverse concrete data formats and common abstract schemas suitable for synchronization), and discuss how the framework can be used to synchronize a variety of specific types of application data by suitable encodings into trees—including sets, records, tuples, relations, and, with some limitations, lists and ordered XML data.

1 Introduction

Optimistic replication is important in settings where weak consistency guarantees are an acceptable price to pay for higher availability and the ability to update data while disconnected. These uncoordinated updates must later be *synchronized* (or *reconciled*) by combining non-conflicting updates and recognizing and dealing with conflicting updates.

Our long-term aim is to develop a generic framework in which high-quality synchronizers for a wide variety of application data formats can be implemented with minimal effort. Our progress toward this goal is embodied in a prototype synchronization framework called Harmony, which focuses on simple edge-labeled trees and offers only limited support for ordered data. An instance of Harmony that synchronizes multiple calendar formats (Palm Datebook, Unix ical, and iCalendar) is in daily use within our group; we have also used Harmony to build a "universal bookmark synchronizer" handling the formats used by several common browsers (Mozilla, Safari, OmniWeb, Internet Explorer 5, and Camino). Other potential Harmony instances include synchronizers for address books, application preference files, geneological data (family trees), file systems, structured documents, drawings, slide presentations, bibliographic databases, and many other forms of semi-structured data.

Some existing synchronizers require *tight coupling* between a synchronization agent and the application programs whose data is being synchronized (so that, for example, the synchronizer can see a complete trace of the operations that the application has performed on each replica of the data, and can propagate changes by undoing and/or replaying operations of the same sorts). Others adopt a *loosely coupled* approach with the goal of synchronizing off-the-shelf applications that were implemented without replication and synchronization in mind. These synchronizers are likely to use a *state-based* approach, in which the synchronizer manipulates application data in an external, on-disk representation such as XML trees. We adopt the latter approach.

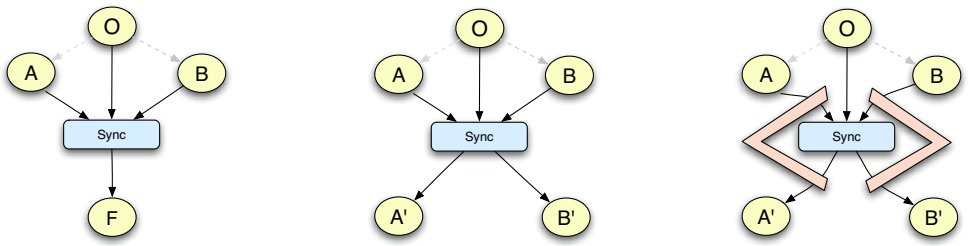
The architecture of Harmony has two major components: (1) a single *synchronization engine* that takes two current replicas and a common ancestor (all three represented as trees) as inputs and yields new replicas in which all non-conflicting changes have been merged, and (2) a collection of *lenses* that are used to prepare data for synchronization, mapping from diverse concrete representations and a common abstract form. Lenses bear the responsibility for "pre-aligning" these abstract trees so that the simple recursive tree-walk performed by the synchronization engine will encounter corresponding substructures at the same moment. This pre-alignment process avoids an n^2 explosion of alignment logic in the heterogeneous setting.

In building Harmony, we have focused a good deal of energy on developing mechanisms that are extremely simple and easy to formalize. Our experience designing and implementing the popular Unison file synchronizer [3, 29] suggests that these properties are prerequisites for a robust implementation and for avoiding behaviors that may surprise users—or even damage their data—in subtle boundary cases.

The main topics of this paper are the core synchronization algorithm, its usefulness for synchronizing various sorts of application data structures, and its precise relation to the rest of the Harmony architecture (the lenses). Our contributions may be summarized as follows:

- We present (in Section 2) the overall architecture through a series of small examples.
- We describe the core synchronization algorithm in detail (Sections 3 and 4). This algorithm, though simple, is carefully crafted to deal sensibly with several basic classes of *conflicts*, including (most interestingly) a notion of "atomicity conflict."
- We give a concise and rigorous statement of the properties that this algorithm (provably) satisfies (Section 4.3).
- The algorithm places some strong demands of completeness (applicability to a sufficiently large domain) on the lenses that may validly be used with it. We characterize these requirements precisely and prove that, when used with suitable lenses, the algorithm is total (Section 5).

  Harmony's domain-specific language for defining lenses, FOCAL, is described in detail in a companion paper [14].
- We show how the algorithm can be used, by varying the encodings performed by lenses, to synchronize a variety of specific types of application data, including sets, records, tuples, and relations (Section 6).



no conflicts with conflicts with lenses

Figure 1: Synchronizer Architecture

- We investigate how ordered data such as lists and XML can be handled in Harmony via similar encodings (Section 7). Our treatment of ordered data is simplistic: inserting or deleting elements anywhere other than at the end of a list may lead to unintuitive conflicts if changes have also occurred in the other replica. However, we argue that it is at least safe, in the sense that unintuitive *propagation* of changes (much worse than spurious conflicts) is avoided.

Sections 8 and 9 discuss related and future work.

Some non-goals of the present paper are worth mentioning explicitly. First, our limited treatment of ordered data is a pragmatic simplification that has allowed us to make progress on other aspects of synchronization. In effect, most of the burden of *aligning* structures—lining up their common parts—is borne by the lenses that are used to prepare the structures for synchronization; the synchronizer itself makes only very local decisions based on equality of labels. This keeps the core algorithm simple and easy to reason about and—more importantly—puts alignment decisions where they belong (in the hands of users, i.e., lens programmers), at the cost of limiting what can be done with ordered data. Ultimately, we hope to extend the core synchronization algorithm to handle ordering, incorporating ideas from known algorithms (see Section 8) for synchronizing various specific forms of ordered data, such as structured text and raw XML documents. It is not immediately clear, however, how these techniques can be adapted to meet our requirements of genericity and heterogeneity. For the same reason, we do not deal here with the problem of recognizing when large substructures have been *moved* in one of the replicas: a move simply shows up as a delete from the old position and a create in the new position. Finally, we focus on the case where just two replicas are to be synchronized. We conjecture that most of the structures we introduce will generalize in a natural way to the more realistic case of multi-replica synchronization, but dealing with many replicas raises additional issues that would overly complicate the discussion at hand.

2 Architecture

Suppose we begin with a tree representing a small phone book:

$$\texttt{O} \quad = \quad \begin{cases} \texttt{Pat} \mapsto \texttt{333-4444} \\ \texttt{Chris} \mapsto \texttt{888-9999} \end{cases}$$

Throughout the paper, we work exclusively with unordered, edge-labeled trees, which we draw sideways to save space. Each curly brace denotes a tree node, and each "$\texttt{X} \mapsto \ldots$" denotes a child labeled X. In running text, we add closing braces to show where trees end. 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., "333-4444" above actually stands for "$\{\texttt{333-4444} \mapsto \{\}\}$."

We now make two replicas of this structure, A and B, and modify one phone number in each of them:

$$\mathtt{A} \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{333\text{-}4444} \\ \mathtt{Chris} \mapsto \mathtt{555\text{-}6666} \end{cases}$$

$$\mathtt{B} \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{111\text{-}2222} \\ \mathtt{Chris} \mapsto \mathtt{888\text{-}9999} \end{cases}$$

Our synchronization tool takes these three structures as inputs and produce an output structure F that reflects the changes made to both replicas:

$$\mathtt{F} \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{111\text{-}2222} \\ \mathtt{Chris} \mapsto \mathtt{555\text{-}6666} \end{cases}$$

The original state O is provided as one of the inputs to the synchronizer so that it can tell which are the updated parts of the replicas. In the simple two-replica case that we are considering in this paper, the archive can be maintained simply by saving a copy of the final merged state F at the end of each synchronization, to use as the O the next time the synchronizer is run. (In a multi-replica system, an appropriate "last shared state" would be calculated in some more complex manner, based on the causal history of the system.) 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 as the left-hand picture in 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.

$$\mathtt{A} \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{123\text{-}4567} \\ \mathtt{Chris} \mapsto \mathtt{555\text{-}6666} \end{cases}$$

$$\mathtt{B} \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{333\text{-}4444} \end{cases}$$

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

1. 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:

   $$\mathtt{F} \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{111\text{-}2222} \\ \mathtt{Chris} \mapsto \mathtt{555\text{-}6666} \end{cases}$$

   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:

   $$\mathtt{A'} \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{123\text{-}4567} \\ \mathtt{Chris} \mapsto \mathtt{555\text{-}6666} \end{cases}$$

   $$\mathtt{B'} \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{123\text{-}4567} \end{cases}$$

   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 8 discusses these trade-offs further. With this refinement, the schematic view of the synchronizer looks like the middle picture in Figure 1.

Our next task is to deal with *heterogeneous* data representations. For example, suppose that our two phone book replicas are stored concretely like this:

$$\mathtt{A} \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{333\text{-}4444} \\ \mathtt{Jo} \mapsto \mathtt{314\text{-}1593} \end{cases}$$

$$\mathtt{B} \;=\; \begin{cases} \mathtt{ID235} \mapsto \begin{cases} \mathtt{FirstName} \mapsto \mathtt{Pat} \\ \mathtt{LastName} \mapsto \mathtt{Sherman} \\ \mathtt{Phone} \mapsto \mathtt{299\text{-}7924} \\ \mathtt{City} \mapsto \mathtt{Sydney} \end{cases} \\ \mathtt{ID923} \mapsto \begin{cases} \mathtt{FirstName} \mapsto \mathtt{Chris} \\ \mathtt{LastName} \mapsto \mathtt{Stephenson} \\ \mathtt{Phone} \mapsto \mathtt{555\text{-}6666} \\ \mathtt{City} \mapsto \mathtt{Qwghlm} \end{cases} \\ \mathtt{ID995} \mapsto \begin{cases} \mathtt{FirstName} \mapsto \mathtt{Alex} \\ \mathtt{LastName} \mapsto \mathtt{Ical} \\ \mathtt{Phone} \mapsto \mathtt{271\text{-}8281} \end{cases} \end{cases}$$

The format of the first is as before. The second has a more complex structure, containing some additional information that, in this example, we are choosing not to synchronize because it cannot be represented in the first replica. (For the sake of the example, we are assuming that the key field is `FirstName` and that the `LastName` field is not synchronized. The fields `FirstName`, `LastName` and `Phone` are required by the concrete format; the others are optional.) Note that we have deleted the record for `Chris` from replica A, added a new record for `Jo` to A, and added a new record for `Alex` to B.

Before we can synchronize these structures, we need to transform them into a common form. In general, both structures may need to be transformed to some common "abstract form" different from either; in this example, we can simply take the abstract schema to be the same as that of A (and O) and transform just B. To transform B to this form, some fields need to be suppressed and some need to be renamed; also, a level of structure (the `IDnnn` edges) needs to be flattened.

$$\mathtt{B} \;=\; \begin{cases} \mathtt{Chris} \mapsto \mathtt{555\text{-}6666} \\ \mathtt{Pat} \mapsto \mathtt{299\text{-}7924} \\ \mathtt{Alex} \mapsto \mathtt{271\text{-}8281} \end{cases}$$

The results of synchronization are identical abstract replicas:

$$\mathtt{A}' \;=\; \mathtt{B}' \;=\; \begin{cases} \mathtt{Pat} \mapsto \mathtt{299\text{-}7924} \\ \mathtt{Jo} \mapsto \mathtt{314\text{-}1593} \\ \mathtt{Alex} \mapsto \mathtt{271\text{-}8281} \end{cases}$$

Of course, the mapping from concrete to abstract structures is only half the story: after synchronization, we need to put our updated replicas back in their original, concrete form. To do this, we need to supply, for each abstraction function, a corresponding *concretion* function that, intuitively, inverts its behavior. (More precisely: the concretion function takes an updated abstract structure and an original concrete structure and returns an updated concrete structure.) We

call these pairs of abstraction and concretion functions *lenses*. The architecture now looks like the right-hand picture in Figure 1. Notice that no lens is applied to O in the picture: we assume that the archive is kept in abstract form.

In the present example, the concretion function maps B′ (together with B) into the following updated version (B″) of B:

$$
\mathtt{B''} \quad = \quad \left\{ \begin{array}{ll}
\mathtt{ID235} \mapsto & \left\{ \begin{array}{l} \mathtt{FirstName} \mapsto \mathtt{Pat} \\ \mathtt{LastName} \mapsto \mathtt{Sherman} \\ \mathtt{Phone} \mapsto \mathtt{299\text{-}7924} \\ \mathtt{City} \mapsto \mathtt{Sydney} \end{array} \right. \\
\mathtt{ID995} \mapsto & \left\{ \begin{array}{l} \mathtt{FirstName} \mapsto \mathtt{Alex} \\ \mathtt{LastName} \mapsto \mathtt{Ical} \\ \mathtt{Phone} \mapsto \mathtt{271\text{-}8281} \end{array} \right. \\
\mathtt{ID999} \mapsto & \left\{ \begin{array}{l} \mathtt{FirstName} \mapsto \mathtt{Jo} \\ \mathtt{LastName} \mapsto \mathtt{UNKNOWN} \\ \mathtt{Phone} \mapsto \mathtt{314\text{-}1593} \end{array} \right.
\end{array} \right.
$$

Note that the IDnnn label and the required LastName field are not available when creating the concrete entry for Jo in B, so the concretion function must make up some values, here ID999 and UNKNOWN. These points give an indication of the numerous subtle details that must be handled in the programming of even fairly simple lenses.

The problem of pushing abstract updates down into concrete structures is an instance of the classical problem of *view update* [8, 4, 13]. In [14] we discuss this aspect of Harmony in more depth. Our aim here, however, is to describe an *architecture* for synchronization in which lenses are used to map heterogeneous concrete formats into common abstract ones prior to synchronization and map updated abstract trees back to updated concrete structures after synchronization. This architecture can be instantiated with any solution (or partial solution) to the view update problem, so long as it satisfies certain constraints, described in Section 5, which guarantee it will "fit properly" with our synchronization algorithm. In our Harmony prototype, we've designed and implemented one particular partial solution, specialized to work with tree-structured data at both the concrete and the abstract level. Our approach is to supply "Harmony programmers" with a domain-specific language for expressing lenses. In this language, every expression denotes a lens and all expressible lenses are guaranteed to unambiguously map modifications to the abstract view to modifications to the underlying concrete view. (More precisely: we can show by construction that all expressible lenses obey a set of simple laws related to Bancilhon and Spyratos's *view update under constant complement* condition [4] and isomorphic to Gottlob et al's *dynamic views* [13]).

3 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 a little more depth. They come in several specific forms, each of which affects the definition of the synchronization 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 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

$$\begin{aligned} \mathtt{O} &= \left\{ \mathtt{Pat} \mapsto \left\{ \begin{array}{l} \mathtt{Phone} \mapsto \mathtt{333\text{-}4444} \\ \mathtt{URL} \mapsto \mathtt{here@there.net} \end{array} \right. \right. \\ \mathtt{A} &= \left\{ \right. \\ \mathtt{B} &= \left\{ \mathtt{Pat} \mapsto \left\{ \begin{array}{l} \mathtt{Phone} \mapsto \mathtt{222\text{-}0000} \\ \mathtt{URL} \mapsto \mathtt{here@there.net} \end{array} \right. \right. \end{aligned}$$

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 $\mathtt{B'} = \{\mathtt{Pat} \mapsto \{\mathtt{Phone} \mapsto \mathtt{222\text{-}0000}\}\}$. 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 path A (here, 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:

$$\begin{aligned} \mathtt{O} &= \left\{ \mathtt{Pat} \mapsto \left\{ \begin{array}{l} \mathtt{Phone} \mapsto \mathtt{333\text{-}4444} \\ \mathtt{URL} \mapsto \mathtt{here@there.net} \end{array} \right. \right. \\ \mathtt{A} &= \left\{ \right. \\ \mathtt{B} &= \left\{ \mathtt{Pat} \mapsto \left\{ \mathtt{Phone} \mapsto \mathtt{333\text{-}4444} \right. \right. \end{aligned}$$

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, this choice would lead to a somewhat more complex specification of the algorithm in the next section, so we have chosen here the more conservative alternative of treating 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:

$$\begin{aligned} \mathtt{O} &= \left\{ \right. \\ \mathtt{A} &= \left\{ \mathtt{Pat} \mapsto \left\{ \mathtt{Phone} \mapsto \mathtt{333\text{-}4444} \right. \right. \\ \mathtt{B} &= \left\{ \mathtt{Pat} \mapsto \left\{ \mathtt{URL} \mapsto \mathtt{here@gone.com} \right. \right. \end{aligned}$$

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

$$\mathtt{A'} = \mathtt{B'} = \left\{ \mathtt{Pat} \mapsto \left\{ \begin{array}{l} \mathtt{Phone} \mapsto \mathtt{333\text{-}4444} \\ \mathtt{URL} \mapsto \mathtt{here@gone.com} \end{array} \right. \right.$$

Formally, in contrast to the delete/delete case, it is slightly *easier* to treat such situations as non-conflicting (treating them as conflicting requires one additional clause in Definition 4.3). However, on pragmatic grounds, 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, the latter class can also be handled by the mechanism of *atomicity conflicts*, which we introduce next. We use @ labels, described below, to explicitly partition the set of create/create situations into those we should treat as conflicts and those we should not.

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 4). 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.)

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

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

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

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

$$\texttt{A}' = \texttt{B}' = \begin{Bmatrix}\texttt{Pat} \mapsto \begin{Bmatrix}\texttt{Phone} \mapsto \begin{Bmatrix}\texttt{111-2222} \mapsto \{ \\ \texttt{987-6543} \mapsto \{\end{Bmatrix}\end{Bmatrix}$$

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.[1] Section 9 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 4.3.) 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

[1] 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.

relationship between t and @. We show in Section 7 how we can use this to control the synchronizer's behavior on complex ordered structures such as lists.

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

$$\begin{array}{lcl} \texttt{O} & = & \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{@ \mapsto \{\texttt{333-4444} \mapsto \{ \\ \texttt{A} & = & \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{@ \mapsto \{\texttt{111-2222} \mapsto \{ \\ \texttt{B} & = & \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{@ \mapsto \{\texttt{987-6543} \mapsto \{ \end{array}$$

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

4 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.[2] We close the section by establishing some key invariants for synchronization of structures involving atomicity.

4.1 Notation

We write N for the set of character strings and T for the set of unordered, edge-labeled trees whose labels are drawn from 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 $\mathsf{dom}(t)$ for the domain of a tree t—i.e. the set of the names of its immediate children.

When $n \notin \mathsf{dom}(t)$, 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 $\mathsf{dom}(\perp) = \emptyset$.

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 $T_{\mathcal{X}}$ for the set of extended trees that may contain $\mathcal{X}$ as a subtree. We write $T_{\perp}$ for the set $T \cup \{\perp\}$ and $T_{\mathcal{X}\perp}$ for the set $T_{\mathcal{X}} \cup \{\perp\}$; we call the latter set *archives*. We define $\mathsf{dom}(\mathcal{X}) = \{n_{\mathcal{X}}\}$, 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:

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

[2] 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 2: Core Synchronization Algorithm

In the proofs we often proceed by induction on the height of a tree. We define $height(\bot) = height(\mathcal{X}) = 0$ and the height of any other tree to be $height(t) = 1 + \mathtt{max}(\{height(t(k)) \mid k \in \mathsf{dom}(t)\})$. 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.

4.2 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 $\bot$, 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 $T_{\mathcal{X}\bot} \times T_\bot \times T_\bot \to T_{\mathcal{X}\bot} \times T_\bot \times T_\bot$.

In the case where A and B already agree (they are both the same tree or both $\bot$), 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.

4.3 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.

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

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

4.2 Lemma: The local equivalence relation is an equivalence.

Proof: The definition is obviously reflexive and symmetric. For transitivity, choose any $t, t', t'' \in T_{\mathcal{X}\perp}$ 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.

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

4.4 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):

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

4.5 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.

4.6 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. □

4.7 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.

4.8 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.

4.9 Theorem: If $\texttt{sync}(o, a, b)$ 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 4.6, the run $(\mathcal{X}, a, b, \mathcal{X}, a, b)$ is safe. We now show that we have $\mathit{conflict}(\mathcal{X}, a, b)$. 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 4.6, the run is safe. We now prove that it is maximal. To this end, we first prove that we have $\mathit{conflict}(o, a, b)$. 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 $\mathit{conflict}(o, a, b)$.

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 4.6.

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

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 $@ \in \mathsf{dom}(a)$ or $@ \in \mathsf{dom}(b)$ (or both), the third case of the definition of $\sim$ cannot apply. Finally, in the case where $@ \in \mathsf{dom}(a) \cap \mathsf{dom}(b)$, 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 4.7, $(o(k), a(k), b(k), o''(k), a''(k), b''(k))$ 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. □

4.4 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.

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

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

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

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

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

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

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

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

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

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

4.11 Definition [Tree Prefix]: The relation $<$ on $T \times T$ is defined as the smallest relation such that:

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

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

4.12 Lemma: Suppose $t \in T$ 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 4.10.

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 $k \in \mathsf{dom}(t)$. 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 $k \in \mathsf{dom}(a')$ and $k \in \mathsf{dom}(b')$.

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

4.13 Lemma: Suppose a and b are atomic trees such that $@$ does not occur in $\mathsf{dom}(a(@))$ or $\mathsf{dom}(b(@))$, and let $(o', a', b') = \mathtt{sync}(o, a, b)$. 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 $\mathsf{dom}(a(@)) = \mathsf{dom}(a \setminus @) = \mathsf{dom}(b \setminus @) = D$. Let $k \in D$, then we have $t(k) < a(k)$ and $t(k) < b(k)$. By Lemma 4.12, we have $t(k) < a'(k)$. This also implies that $a'(k) \neq \bot$, hence $k \in \mathsf{dom}(a')$. Hence we have $\mathsf{dom}(a(@)) \subseteq \mathsf{dom}(a' \setminus @)$. Let k be a name that is neither $@$ nor in $\mathsf{dom}(a(@))$, then $k \notin \mathsf{dom}(a)$ and $k \notin \mathsf{dom}(b)$, hence (by the first case of the synchronization algorithm with $a(k) = b(k) = \bot$), we have $k \notin \mathsf{dom}(a')$. Hence we have $\mathsf{dom}(a(@)) = \mathsf{dom}(a' \setminus @)$ and for all $k \in \mathsf{dom}(a(@))$, $(a(@))(k) < (a' \setminus @)(k)$, thus $a'(@) = a(@) < a' \setminus @$. □

4.14 Lemma: Suppose a and b are atomic and $(o', a', b') = \mathtt{sync}(o, a, b)$. 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 4.13 and 4.14 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 $a(@) = b(@)$.

5 Putting the Pieces Together

As our approach is parameterized by a lens language that maps concrete representations into abstract trees and maps back synchronized abstract trees into concrete format, we need a way of guaranteeing that a successful run of the synchronization algorithm produces trees that can be handled by the lens language.

A *lens* l from some set C of concrete structures to a set A of abstract trees comprises a *get* function from C to A and a *put* function from $A \times C$ to C. (See [14] for more details.) Write $A_{\mathcal{X}}$ for the subset of $T_{\mathcal{X}}$ formed by taking a tree from A and replacing any number of subtrees by $\mathcal{X}$. Write $A_\perp$ for the set $A \cup \{\perp\}$ and $A_{\mathcal{X}\perp}$ for $A_{\mathcal{X}} \cup \{\perp\}$.

5.1 Definition: The set of trees A is said to be *closed under synchronization* iff, for all $o \in A_{\mathcal{X}\perp}$, all $a, b \in A_\perp$, and $(o', a', b') = \texttt{sync}(o, a, b)$, we have $o' \in A_{\mathcal{X}\perp}$ and $a', b' \in A_\perp$.

Now, given two concrete sets C_1 and C_2, a common abstract set A, and lenses l_1 from C_1 to A and l_2 from C_2 to A, if A is closed under synchronization, then we can rest assured that the whole process of applying lenses to concrete replicas, synchronizing, and using the lenses to put the results back in concrete form will always succeed.

The closure of a given set A under synchronization is often obvious, but this is not always the case. Section 7 explores some interesting examples.

6 The Art of Alignment

When programming specific synchronizers using Harmony, one key issue consists of choosing a schema for the abstract trees so that our straightforward, tree-walking-by-names synchronization algorithm aligns data structures correctly. In this section we introduce synchronization-friendly abstract schemas for some commonly encountered data structures. We elide the details of how to actually construct lenses performing the mappings between diverse concrete application formats and the schemas sketched here; Hamony's domain-specific lens programming language, FOCAL [14], is one possibility, but the issues here are orthogonal to how lenses are implemented.

Sets

We first consider sets of unstructured values. In this case, alignment only consists of deciding whether a given value is present in both sets.

Since an unordered, edge-labeled tree may be viewed as a set of children bearing pairwise distinct names, a synchronization-friendly schema for a set of values simply is a tree where each value is encoded in the name of a child: the set with elements `value1`...`valuen` is encoded as the tree $\{\texttt{value1}, \ldots, \texttt{valuen}\}$.

Records

A record is a data structure where *fields* point to data, and fields should be aligned according to their names. In order to align the data under each field, one simply needs to represent the record as a bush, with each child bearing the name of a field and pointing to the data: $\{\texttt{field1} \mapsto t_1, \ldots \texttt{fieldn} \mapsto t_n\}$.

Tuples

A tuple may be considered as a record indexed by position. Hence the tuple (t_1, t_2, t_3) may be encoded as the tree $\{\texttt{1} \mapsto t_1; \texttt{2} \mapsto t_2; \texttt{3} \mapsto t_3\}$.

K	A	B
k_1	a_1	b_1
k_2	a_2	b_2

can be represented as:

$$\begin{cases} k_1 \mapsto \begin{cases} A \mapsto a_1 \\ B \mapsto b_1 \end{cases} \\ k_2 \mapsto \begin{cases} A \mapsto a_2 \\ B \mapsto b_2 \end{cases} \end{cases}$$

Figure 3: Representing relations as trees

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 should be associated to which tuple of each replica. Consider for instance the following relations:

$$O = \{(\texttt{Pat}, \texttt{333-4444}); (\texttt{Chris}, \texttt{888-9999})\}$$
$$A = \{(\texttt{Pat}, \texttt{111-2222}); (\texttt{Chris}, \texttt{888-9999})\}$$
$$B = \{(\texttt{Pat}, \texttt{123-4567}); (\texttt{Jo}, \texttt{888-9999})\}$$

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), then the result of synchronization is the tree:

$$\begin{cases} \texttt{Pat:111-2222} \\ \texttt{Pat:123-4567} \\ \texttt{Jo:888-9999} \end{cases}$$

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 two independent non-conflicting additions.

A schema-based solution to this issue consists of choosing 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.

A satisfying representation for our simple example would thus be (for instance for O):

$$\begin{cases} \texttt{Pat} \mapsto \left\{ \texttt{Phone} \mapsto \left\{ @ \mapsto \texttt{333-4444} \right. \right. \\ \texttt{Chris} \mapsto \left\{ \texttt{Phone} \mapsto \left\{ @ \mapsto \texttt{888-9999} \right. \right. \end{cases}$$

(The phone numbers are atomic to make them values. The synchronization of O, A, and B hence leads to an atomic conflict.)

Note that this schema fails if the value of the key is changed, as this change would in fact be interpreted as a complete deletion and addition of a record. This issue is easily circumvented when every tuple contains a non-modifiable key, such as a unique identifier.

7 Ordered Structures

The final technical section discusses the treatment of ordered structures such as lists, text, and ordered XML data. A key observation is that ordered structures come in two distinct flavors: one where the number of elements is variable but where the absolute position of elements is what matters to the application or user whose data we are synchronizing—we call these *extensible tuples*—and a richer one—full-blown *lists*—where it is the relative position of the elements that is significant. For extensible tuples, we can give an abstract schema for which Harmony's synchronization algorithm behaves very intuitively. For lists, we must be content with behavior that is *safe*—i.e., the synchronizer propagates changes correctly in a limited range of situations (such as when only one replica has been modified), and in all other situations it signals a conflict (rather than producing malformed or counter-intuitive results).

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

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

—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

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

$$A = \{$$

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

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

$$O' = \{\mathtt{3} \mapsto \mathcal{X}$$

$$A' = \{$$

$$B' = \{\mathtt{3} \mapsto \mathtt{c}$$

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:

$$\begin{cases} \mathtt{*h} \mapsto \mathtt{t}_1 \\ \mathtt{*t} \mapsto \begin{cases} \mathtt{*h} \mapsto \mathtt{t}_2 \\ \mathtt{*t} \mapsto \begin{cases} \ldots \mapsto \begin{cases} \mathtt{*h} \mapsto \mathtt{t}_n \\ \mathtt{*t} \end{cases} \end{cases} \end{cases} \end{cases}$$

That is, a tree t represents an extensible tuple iff it is empty or has exactly two children, one named `*h` and another named `*t`, with $t($`*t`$)$ 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:

$$O = \begin{cases} \mathtt{*h} \mapsto \{\mathtt{Pat} \mapsto \mathtt{333\text{-}4444} \\ \mathtt{*t} \mapsto \{ \end{cases}$$

$$A = \{$$

$$B = \begin{cases} \mathtt{*h} \mapsto \{\mathtt{Pat} \mapsto \mathtt{111\text{-}2222} \\ \mathtt{*t} \mapsto \{ \end{cases}$$

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):

$$B' = \left\{ \texttt{*h} \mapsto \left\{ \texttt{Pat} \mapsto \texttt{111-2222} \right.\right.$$

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 $\{@ \mapsto \{\texttt{*h}, \texttt{*t}\}\}$ for internal nodes (cons cells). The synchronization now results in an atomic conflict at the root of the tree.

$$O = \begin{cases} @ \mapsto \begin{cases} \texttt{*h} \\ \texttt{*t} \end{cases} \\ \texttt{*h} \mapsto \left\{ \texttt{Pat} \mapsto \texttt{333-4444} \right. \\ \texttt{*t} \mapsto \left\{ @ \mapsto \{ \right. \end{cases}$$

$$A = \left\{ @ \mapsto \{ \right.$$

$$B = \begin{cases} @ \mapsto \begin{cases} \texttt{*h} \\ \texttt{*t} \end{cases} \\ \texttt{*h} \mapsto \left\{ \texttt{Pat} \mapsto \texttt{111-2222} \right. \\ \texttt{*t} \mapsto \left\{ @ \mapsto \{ \right. \end{cases}$$

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.

7.1 Proposition: Let $o \in T_{\mathcal{X}\perp}$, 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 4.14, 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 4.13 $a'(@) = a(@) = b(@)$, 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 $k \in \texttt{dom}(a)$ or $k \in \texttt{dom}(b)$. 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:

$$O = \left\{ \begin{array}{l} @ \mapsto \left\{ \begin{array}{l} \texttt{*h} \mapsto \texttt{Pat} \\ \texttt{*t} \end{array} \right. \\ \texttt{*h} \mapsto \left\{ \texttt{Pat} \mapsto \texttt{333} \right. \\ \texttt{*t} \mapsto \left\{ \begin{array}{l} @ \mapsto \left\{ \begin{array}{l} \texttt{*h} \mapsto \texttt{Chris} \\ \texttt{*t} \end{array} \right. \\ \texttt{*h} \mapsto \left\{ \texttt{Chris} \mapsto \texttt{888} \right. \\ \texttt{*t} \mapsto \left\{ @ \mapsto \{ \right. \end{array} \right. \end{array} \right. \end{array} \right.$$

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.

7.2 Proposition: Let $o \in T_{\mathcal{X}\perp}$, 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.

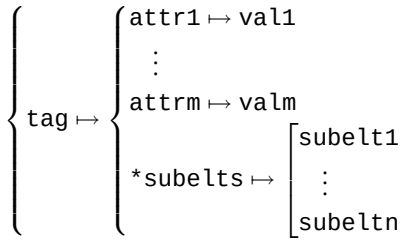
Proof: A list encoding t is well formed if either t is the empty tree or else (i) t has three children @, `*h`, and `*t`, (ii) $\texttt{dom}(t(@)) = \{\texttt{*h}, \texttt{*t}\}$, (iii) $t(@)(*h) < t(\texttt{*h})$, and (iv) $t(\texttt{*t})$ is itself a well-formed list. Without loss of generality, suppose 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 4.14, 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 4.13, either $a'(@) = a(@)$ or $a'(@) = b(@)$, both of which are already known to be well-formed subtrees of @ under the encoding of lists. Moreover, $a'(@) < a' \backslash @$ 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 $k \in \texttt{dom}(a)$ or $k \in \texttt{dom}(b)$. 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. □

XML

Building on the 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:



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.

8 Related Work

The Harmony framework combines a core synchronization component and a view update component for dealing with heterogeneity. The view update language is described in [14], and we refer the reader there for related work. In this section, we focus mainly on related work on optimistic replication and synchronization.

Harmony is an instance of a large class of systems that perform *optimistic replication*. The reader is directed to an excellent article by Saito and Shapiro [36] surveying the area. 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 [10]), (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. [7, 28, 38, 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 by expressing syntactic/static and semantic/dynamic ordering constraints between operations. Molli et al [24, 17, 25], have also implemented a generic operation-based reconciler, using operational transformation. It is parameterized by writing transformation functions for all operations, satisfying a set of formal conditions. Like Harmony they formally specify the behavior of their system.

Bengal [11] 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 intended to be 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 7.

File system synchronizers (such as [37, 27, 15, 35, 3, 33]) and PDA synchronizers (such as Palm HotSync), are not generic, but they do generally share Harmony's state-based approach.

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 resolve conflicts. This is impossible to do in the general case without discarding updates.
- 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 [15], Clique [35], Bengal [11], 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 4 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

Unsurprisingly, given our goal of reconciling heterogeneous data sources, we find 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 [12, 1, 16, 39]. If we consider the (non-trivial) problem of augmenting a data integration system with view update (another well-studied area—see [14] for a survey), then the result can be used to implement an optimistic replication system that can reconcile conflicts between heterogeneous data sources[3]. 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, 5, 9, 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 [6], and FCDP [19].

[3]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.

9 Future Work

The Harmony prototype currently supports several synchronizer instances. One of these is in daily use within our group for synchronizing small (hundreds of records) calendar files in various formats. Several others are under development. We are also working hard on user interface issues.

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. We have a preliminary design for this extension that seems promising.

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 [6], 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. We also wonder whether at least parts of the framework could be adapted to a relational setting.

Finally, we have observed that the create/create and atomicity conflicts discussed in Section 3 can both be viewed as specific instances of a more general notion of *schema* (or *type*) *conflicts*. In the final example in that section, for instance, the atomicity edge encodes the constraint that nodes representing "values" should be single-valued, in the sense that the result of synchronizing two values will always be two values (i.e., either two copies of the same value, or a conflict). If we could make the synchronizer aware of the schema of the abstract structures, then we would have a more direct, and more powerful, way of avoiding these situations and many others.

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. Jonathan Moore was our collaborator on the initial design of the FOCAL language and contributed ideas and code to many parts of the Harmony system. Owen Gunden took the system yet further, designing and implementing an "optometrist" module for automatically choosing lenses based on file types and implementing several demo applications. Our current work with Nate Foster (on FOCAL) and Sanjeev Khanna (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*.

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] F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 6(4):557–575, 1981.

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

[6] 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, Edinburgh, Scotland, U.K., 1996.

[7] 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.

[8] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *TODS*, 7(3):381–416, September 1982.

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

[10] 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.

[11] 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.

[12] 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.

[13] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *TODS*, 13(4):486–524, 1988.

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

[15] 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.

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

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

[37] 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.

[38] 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.

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