HAL
open science

Exploiting Schemas in Data Synchronization

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

To cite this version:

J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, Alan Schmitt. Exploiting Schemas in Data Synchronization. Journal of Computer and System Sciences, 2007, 73 (4), pp.669-689. 10.1016/j.jcss.2006.10.024 . inria-00483199

HAL Id: inria-00483199

https://inria.hal.science/inria-00483199v1

Submitted on 12 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Exploiting Schemas in Data Synchronization [1]

J. Nathan Foster [a] Michael B. Greenwald [b],
Christian Kirkegaard [c], Benjamin C. Pierce [a] and Alan Schmitt [d]

[a] *University of Pennsylvania, Philadelphia, PA 19104 USA*

[b] *Bell Labs, Lucent Technologies, Murray Hill, NJ 07974 USA*

[c] *BRICS, University of Aarhus, DK-8000 Aarhus C, Denmark*

[d] *INRIA Rhône-Alpes, 38330 Montbonnot, France*

Preprint submitted to Elsevier Science *12 October 2006*

Abstract

Increased reliance on optimistic data replication has led to burgeoning interest in tools and frameworks for *synchronizing* disconnected updates to replicated data. But good data synchronizers are challenging both to specify and to build.

We have implemented a generic synchronization framework, called Harmony, that can be used to build state-based synchronizers for a wide variety of tree-structured data formats. A novel feature of this framework is that the synchronization process—in particular, the recognition of conflicts—is driven by the schema of the structures being synchronized.

We formalize Harmony's synchronization algorithm, prove that it obeys a simple and intuitive specification, and illustrate, using simple address books as a case study, how it can be used to synchronize trees representing a variety of specific forms of application data, including sets, records, tuples, and relations.

Key words: synchronization, optimistic reconciliation, XML, Harmony

[1] Extended version of a paper originally presented at the Symposium on Database Programming Languages (DBPL) in Trondheim, Norway, August 2005.

1 Introduction

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

Our long-term goal is to develop a generic framework that can be used to build high-quality synchronizers, with minimal effort, for a wide variety of application data formats. As a step toward this goal, we have designed and built a prototype synchronization framework called Harmony, focusing on the important special cases of unordered and rigidly ordered data, including sets, relations, tuples, records, feature trees, etc.; the prototype also includes preliminary support for list-structured data such as structured documents, but both the theory and the implementation are less advanced. We have used Harmony to build synchronizers for multiple calendar formats (Palm Datebook, Unix ical, and iCalendar), bookmarks (handling the formats used by several common browsers, including Mozilla, Safari, and Internet Explorer), address books, application preference files, drawings, and bibliographic databases; other applications are construction.

The Harmony system has two main components: (1) a domain-specific programming language for writing *lenses*—bi-directional transformations on trees—which we use to convert low-level and possibly heterogeneous concrete data formats into a common high-level *synchronization schema*, and (2) a generic synchronization algorithm, whose behavior is controlled by the synchronization schema.

The synchronization schema actually guides Harmony's behavior in two ways. First, by choosing an appropriate schema and the lenses that transform concrete structures into this form and back, users of Harmony can control the *alignment* of the information being synchronized: the same concrete format might be transformed to different synchronization schemas (for example, making different choices of keys) to yield quite different synchronization semantics; this process is illustrated in Section 6. Second, during synchronization, the synchronization schema is used to identify *conflicts*—situations where changes in one replica must *not* be propagated to the other because the resulting combined structure would be ill-formed. To our knowledge, Harmony is the first state-based synchronizer to preserve structural invariants in this way. (By contrast, *operation-based* synchronization frameworks preserve application invariants by working in terms of a high-level, application-specific algebra of operations rather than directly manipulating replicated data; see Section 8.)

Harmony's language for lenses has been described in detail elsewhere [1]; in the present paper, our focus is on the synchronization algorithm and the way it uses schema information. The basic behavior of this algorithm is simple: we try to propagate changes from each replica to the other, validate the resulting trees according to the expected schema, and signal a conflict if validation fails. However, the details are somewhat subtle: there may be many changes to propagate from each replica to the others, leading to many possible choices of *where* to signal conflicts (i.e., which subset of the changes to propagate). To ensure progress, we want synchronization to propagate as many changes as possible while respecting the schema; at the same time, to avoid surprising users, we need the results of synchronization to be predictable; for example, small variations in the inputs should not produce large variations in the set of changes that are propagated. A natural way of combining these design constraints is to demand that the results of synchronization be *maximal*: if there is *any* way to validly propagate a given modification from one replica to the other, then that change *must* be propagated. Our main technical contribution is a simple one-pass, recursive tree-walking algorithm that does indeed yield results that are maximal in this sense (and hence also unique) for schemas satisfying a locality constraint called *path consistency* (a semantic variant of the *consistent element declaration* condition in W3C Schema).

After establishing some notation in Section 2, we explore the design space further, beginning in Section 3 with some simple synchronization examples. Section 4 focuses on difficulties that arise in a schema-aware algorithm. Section 5 presents the algorithm itself. (We defer the formal specification to Appendix A.) Section 6 illustrates the behavior of the algorithm using a simple address book schema. Section 7 explores the additional challenge of synchronizing list-structured data and proposes a modest extension to the algorithm for this case. Related and future work are discussed in Sections 8 and 9.

2 Data Model

Internally, Harmony manipulates data in an extremely simple form: unordered, edge-labeled trees; richer external formats such as XML are encoded as unordered trees. We chose this simple data model on pragmatic grounds: experience shows that the reduction in the overall complexity of the Harmony system far outweighs the cost of manipulating ordered data in encoded form (see [1]). We write $\mathcal{N}$ for the set of character strings and $\mathcal{T}$ for the set of unordered, edge-labeled trees whose labels are drawn from $\mathcal{N}$ and where labels of the immediate children of nodes are pairwise distinct. We draw trees sideways to save space: in text, each pair of curly braces denotes a tree node, and each "`X`$\mapsto$..." denotes a child labeled `X`—e.g., {`Pat`$\mapsto$`111-1111`, `Chris`$\mapsto$`222-2222`}. To avoid clutter, when an edge leads to an empty tree, we omit the braces,

the $\mapsto$ symbol, and the final childless node—e.g., "`111-1111`" above actually stands for "$\{$`111-1111`$\mapsto\{\}\}$."

A tree can be viewed as a partial function from names to trees; we write $t(n)$ for the immediate subtree of t labeled with the name n and $\mathsf{dom}(t)$ for its domain—i.e. the set of names of its children. The concatenation operator, $\cdot$, is only defined for trees with disjoint domains; $t \cdot t'$ is the tree mapping n to $t(n)$ for $n \in \mathsf{dom}(t)$ and to $t'(n)$ for $n \in \mathsf{dom}(t')$. When $n \notin \mathsf{dom}(t)$, we define $t(n)$ to be $\perp$, the "missing tree." By convention, we take $\mathsf{dom}(\perp) = \emptyset$. To represent conflicts during synchronization, we further enrich the set of trees with a special pseudo-tree $\mathcal{X}$, pronounced "conflict." If S is a set of ordinary trees (i.e., $\perp \notin S$ and $\mathcal{X}$ is not a subtree of any tree in S), we write $S_\perp$ for $S \cup \{\perp\}$, and $S_{\mathcal{X}\perp}$ for the set obtained from $S_\perp$ by allowing arbitrary subtrees to be replaced by $\mathcal{X}$. We prove many properties by induction on the height of trees. We define $\mathsf{height}(\perp) = \mathsf{height}(\mathcal{X}) = 0$ and the height of an ordinary tree as $\mathsf{height}(t) = 1 + \mathtt{max}(\{\mathsf{height}(t(k)) \mid k \in \mathsf{dom}(t)\})$.

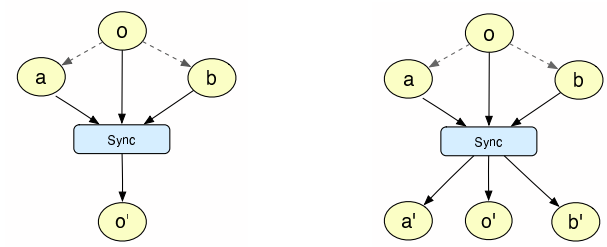
A *path* is a sequence of names. We write $\bullet$ for the empty path and p/q for the concatenation of paths p and q; the set of all paths is written $\mathcal{P}$. The *projection* of t along a path p, written $t(p)$, is defined as follows: (1) $t(\bullet) = t$; (2) $t(n/p) = (t(n))(p)$ if $t \neq \mathcal{X}$ and $n \in \mathsf{dom}(t)$; (3) $t(n/p) = \perp$ if $t \neq \mathcal{X}$ and $n \notin \mathsf{dom}(t)$; and (4) $t(p) = \mathcal{X}$ if $t = \mathcal{X}$. A tree is strictly *included* in another tree, written $t \sqsubset t'$, iff t and t' are different trees and any missing or conflicting path in t' is missing in t. Formally, $\sqsubset$ is the binary relation on $\mathcal{T}_{\mathcal{X}\perp} \times \mathcal{T}_{\mathcal{X}\perp}$ such that $t \sqsubset t'$ iff $t \neq t'$ and for every $p \in \mathcal{P}$ if $t'(p) = \perp$ or $t'(p) = \mathcal{X}$ then $t(p) = \perp$.

Our algorithm is formulated using a semantic notion of *schemas*—a schema S is a set of ordinary trees $S \subseteq \mathcal{T}$. In Section 6 we define a particular syntax for writing down schemas, which is used in the examples and in our prototype implementation; however, our core algorithm does not rely on this particular notion of schema.

3 Basics

Harmony's synchronization algorithm takes two replicas $a, b \in \mathcal{T}_\perp$ and a common ancestor $o \in \mathcal{T}_{\mathcal{X}\perp}$ and yields new replicas in which all non-conflicting updates are merged.[2] The missing tree, $\perp$, represents replicas that have been

[2] We focus on the two-replica case. Our algorithm generalizes straightforwardly to n simultaneously connected replicas, but the more realistic case where only a subset of the replicas may be connected at any given moment poses additional challenges. Some progress in this area is reported in [2].



no conflicts with conflicts

Fig. 1. Synchronizer architecture

completely deleted (note that $\perp$ is different from the empty tree, $\{\}$); $\mathcal{X}$ marks paths in o that a previous synchronization run left in conflict.

Suppose, for example, that we have a tree o representing a phone book and that we make two replicas of this structure, a and b, and separately modify one phone number in each:

$$o = \{\texttt{Pat} \mapsto \texttt{111-1111},\ \texttt{Chris} \mapsto \texttt{222-2222}\}$$
$$a = \{\texttt{Pat} \mapsto \texttt{111-1111},\ \texttt{Chris} \mapsto \texttt{888-8888}\}$$
$$b = \{\texttt{Pat} \mapsto \texttt{999-9999},\ \texttt{Chris} \mapsto \texttt{222-2222}\}$$

Synchronization takes these structures and produces a structure

$$o' = a' = b' = \{\texttt{Pat} \mapsto \texttt{999-9999},\ \texttt{Chris} \mapsto \texttt{888-8888}\}$$

that reflects all the changes in a and b with respect to o. We cache this final merged state, o', at the end of each synchronization, to use as the o input for the next synchronization.[3] Schematically, the synchronizer may be visualized like the diagram on the left of Figure 1.

In the remainder of this section, we describe the fundamental choices in Harmony's architecture and discuss the relative advantages and disadvantages of these choices, compared with alternatives found in other synchronization technologies.

Loose Coupling Harmony is a *state-based* synchronizer: only the current states of the replicas (plus the remembered common ancestor state o) are supplied as inputs; the precise sequences of operations that produced a and b from o are not available to the synchronizer. The reason for this choice is that Harmony is designed to require only *loose coupling* with applications: it manipulates application data in external, on-disk representations such as XML trees. The advantage of the loosely coupled approach is that we can

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

use Harmony to synchronize off-the-shelf applications that were implemented without replication in mind. By contrast, synchronizers that work with the traces of operations that the application has performed on each replica and that propagate changes by undoing and/or replaying operations require tight coupling between the synchronizer and application programs.

Conflicts and Persistence During synchronization, 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 as in the example above, we change both Pat's and Chris's phone numbers in a while, in b, we delete the record for Chris entirely, yielding replicas $a = \{\texttt{Pat} \mapsto \texttt{123-4567},\ \texttt{Chris} \mapsto \texttt{888-8888}\}$ and $b = \{\texttt{Pat} \mapsto \texttt{111-1111}\}$. Clearly, there is no single phone book that incorporates both changes to Chris's phone number. Formally, we have a *delete/create conflict*—the subtree along the path `Chris` was deleted in b and a tree was created in a along the path `Chris/888-8888`. At this point, we must choose between two evils. On one hand, we can weaken users' expectations of the *persistence* of their changes to the replicas—i.e., we can decline to promise that synchronization will never back out changes that the user has made to either replica. For example, here, we might back out the deletion of Chris, yielding $a' = b' = \{\texttt{Pat} \mapsto \texttt{123-4567},\ \texttt{Chris} \mapsto \texttt{888-8888}\}$. The user would then be notified of the lost changes and given the opportunity to re-apply them if desired. Alternatively, we can keep persistence and instead give up *convergence*—i.e., we can allow the replicas to remain different after synchronization, propagating just the non-conflicting change to Pat's phone number and leaving the conflicting information about Chris untouched in each replica— $a' = \{\texttt{Pat} \mapsto \texttt{123-4567},\ \texttt{Chris} \mapsto \texttt{888-8888}\}$ and $b' = \{\texttt{Pat} \mapsto \texttt{123-4567}\}$ —and notify the user of the conflict.[4]

In Harmony, we have chosen to favor persistence because it is easier to ensure that unsupervised reconciliations are safe. (Unsupervised reconciliations are extremely desirable from the point of view of system administration, facilitating automatic reconciliation before disconnection or upon re-connection to a network, via nightly scripts, etc.) 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. By contrast, 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

[4] An industrial-strength synchronization tool will not only report the conflict, but also assist in bringing the replicas back into agreement by providing graphical views of the differences, applying special heuristics, etc. We omit discussion of these aspects of synchronization, focusing on the synchronizer's basic, "unattended" behavior.

causes subsequent processing to fail. For a more detailed survey of convergent systems, see Section 8.

With this refinement, the schematic view of the synchronizer looks like the diagram on the right side of Figure 1.

Local Alignment Another fundamental consideration in the design of any synchronizer is *alignment*—i.e., the mechanism that identifies which parts of each replica represent "the same information" and should be synchronized with each other. Synchronization algorithms can be broadly grouped into two categories, according to whether they make alignment decisions *locally* or *globally*. Synchronizers that use global heuristics for alignment—e.g., the popular Unix tool `diff3`, Lindholm's 3DM [3], the work of Chawathe et al [4], and FCDP [5]—make a "best guess" about what operations the user performed on the replicas by comparing the *entire* current states with the last common state. This works well in many cases (where the best guess is clear), but in boundary cases these algorithms can make surprising alignment decisions. To avoid these issues, our algorithm employs a simple, local alignment strategy that associates the subtrees under children with the same name with each other. The behavior of this scheme should be easy for users to understand and predict. The cost of operating *completely* locally is that Harmony's ability to deal with ordered data is limited, as we discuss in Section 7.

Lenses The local alignment scheme described above works well when the replicas are represented in a format that naturally exposes the structure of the data being synchronized. For example, if the replicas represent address books, then a good representation is as a bush where an appropriate *key field*, providing access to each contact, appears at the root level.

$$\left\{\begin{array}{l}\texttt{92373}\mapsto\left\{\texttt{name}\mapsto\left\{\begin{array}{l}\texttt{first}\mapsto\texttt{Megan},\\ \texttt{last}\mapsto\texttt{Smith}\end{array}\right\},\texttt{home}\mapsto\texttt{555-6666}\right\},\\ \texttt{92374}\mapsto\left\{\texttt{name}\mapsto\left\{\begin{array}{l}\texttt{first}\mapsto\texttt{Pat},\\ \texttt{last}\mapsto\texttt{Jones}\end{array}\right\},\texttt{home}\mapsto\texttt{555-2222}\right\}\end{array}\right\}$$

Using the local alignment scheme described above, the effect during synchronization will be that entries from the two replicas with the same key are synchronized with each other. Alternatively, if no key is available, we might instead synthesize one by lifting information out of each record—e.g., by concatenating the `name` data and using it as the top-level key field:

$$\left\{\begin{array}{l}\texttt{Megan:Smith}\mapsto\left\{\texttt{home}\mapsto\texttt{555-6666}\right\},\\ \texttt{Pat:Jones}\mapsto\left\{\texttt{home}\mapsto\texttt{555-2222}\right\}\end{array}\right\}$$

It is unlikely, however, that the address book will be represented concretely (e.g., on disk) in either of these formats. To bridge this gap between actual, application-determined concrete formats and more synchronization-friendly

representations, the Harmony system includes a domain-specific language for writing bi-directional transformations, which we call *lenses* [1]. By passing each replica through a lens, we can transform the replicas from concrete formats into appropriately "pre-aligned" forms. After synchronization, our language guarantees that the updated replicas are transformed back into the appropriate concrete formats using the other side of the same lens (i.e., lenses can be thought of as *view update translators* [6]). Lenses also facilitate synchronization of heterogeneous formats. Since each replica is passed through a lens both before and after synchronization, the replicas need not be represented in the same format. We can apply a different lens to each replica to transform the disparate concrete representations into the same format for synchronization.

4 The Role of Schemas

To formalize our intuitions about what a user may reasonably expect from a synchronizer, we impose two core requirements, which we call *safety* and *maximality*. We describe them informally in this section; Appendix A gives precise definitions.

The safety requirement encompasses four basic sanity checks: (1) The synchronizer must not "back out" any changes made at a replica since the last synchronization (because we favor persistence over convergence); (2) it must only copy data between replicas, never "making up" content; (3) it must halt at conflicting paths, leaving the replicas untouched below; (4) it must produce results that belong to the same schema as the originals.

Of course, safety alone is too weak: an algorithm that returns both replicas unchanged would be trivially safe! We therefore say that a safe run is *maximal* just in case it propagates all the changes of every other safe run. Our formal specification, given in Appendix A, demands that every run be maximal.

Our algorithm, unlike other state-based synchronizers, is designed to preserve structural invariants. As an example of how such invariants can be broken, consider a run of the algorithm as sketched above, where the inputs are as follows (we revert to the fully explicit notation for trees here, to remind the reader that "leaf values" are labels that lead to an empty subtree):

$$o = \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{333-4444} \mapsto \{\}\}\}\}$$
$$a = \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{111-2222} \mapsto \{\}\}\}\}$$
$$b = \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{987-6543} \mapsto \{\}\}\}\}$$

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'; similarly, we can propagate the creation of `987-6543` to a', yielding

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

But this behavior is wrong. Pat's phone number was *changed* in different ways in the two replicas. If the phonebook schema only allows a single number per person, then the new replica is not valid; the desired behavior is a conflict. We avoid these situations by providing the schema itself as an input to the synchronizer. The synchronizer signals a conflict (leaving the replicas unchanged) if merging the changes along a particular path yields an ill-formed structure.

Locality and Schemas

A simple way to ensure that the results produced by the synchronization algorithm are valid would be to check for schema compliance at the end of each run, halting with a conflict if the results do not belong to the schema. This approach is not satisfying, however, since it discards every modification even if only a small part of the result is not compliant. For example, consider synchronizing replicas of a large address book where only one entry does not belong to the schema after synchronization. Using the approach described above, the algorithm halts with a conflict on the *entire* address book because that single entry is ill-formed and declines to propagate safe updates to any other entries. As this clearly conflicts with our goal of maximality, we have taken a more local approach to schema compliance, which entails some restrictions to the schemas that may be used. To motivate these restrictions, consider the following schema: $\{\{\}, \{\texttt{n} \mapsto \texttt{x}, \texttt{m} \mapsto \texttt{x}\}, \{\texttt{n} \mapsto \texttt{y}, \texttt{m} \mapsto \texttt{y}, \}, \{\texttt{n} \mapsto \{\texttt{x}, \texttt{y}\}, \texttt{m} \mapsto \texttt{y}\}, \{\texttt{n} \mapsto \texttt{x}, \texttt{m} \mapsto \{\texttt{x}, \texttt{y}\}\}\}$. This schema expresses a non-local invariant: at most one of `m` and `n` has $\{\texttt{x}, \texttt{y}\}$ as a subtree. Now, consider synchronizing two replicas belonging to this schema with respect to an empty archive, $o = \{\}$, with $a = \{\texttt{n} \mapsto \texttt{x},\ \texttt{m} \mapsto \texttt{x}\}$ and $b = \{\texttt{n} \mapsto \texttt{y},\ \texttt{m} \mapsto \texttt{y}\}$. An algorithm that aligns each replica by name will recursively synchronize the associated subtrees below `n` and `m`. However, it is not clear what *schema* to use for these recursive calls, because the set of trees that can validly appear under `n` depends on the subtree under `m` and vice versa. We might try the schema that consists of all the trees that can appear under `n` (and `m`): $\{\texttt{x},\ \texttt{y}, \{\texttt{x},\ \texttt{y}\}\}$.With this schema, the synchronizer computes the tree $\{\texttt{x},\ \texttt{y}\}$ for both `n` and `m`, reflecting the fact that `x` and `y` were both added under `n` and `m`. However, these trees cannot be assembled into a well-formed tree: $\{\texttt{n} \mapsto \{\texttt{x}, \texttt{y}\},\ \texttt{m} \mapsto \{\texttt{x}, \texttt{y}\}\}$ does not belong to the schema. The "most synchronized" well-formed results are $a' = \{\texttt{n} \mapsto \texttt{x},\ \texttt{m} \mapsto \{\texttt{x},\ \texttt{y}\}\}$ and $b' = \{\texttt{n} \mapsto \{\texttt{x},\ \texttt{y}\},\ \texttt{m} \mapsto \texttt{y}\}$, but there does not seem to be any way to find them efficiently. The global invariant expressed by this schema—that at most one of `n` or `m` may have $\{\texttt{x},\ \texttt{y}\}$ as a subtree—cannot

easily be preserved by a local algorithm.

To avoid such situations, we impose a restriction on schemas, *path consistency*, that is analogous to the restriction on tree grammars embodied by W3C Schema. Intuitively, a schema is path consistent if any subtree that appears at some path in one tree can validly be "transplanted" to the same location in any other tree in the schema. This restriction ensures that the sub-schema used to synchronize each child is consistent across the schema; i.e., the set of trees that may validly appear under a child only depends on the path to the node and not the presence or absence of other information elsewhere in the tree.

To define path consistency precisely, we need a little new notation. First, the notion of projection at a path is extended pointwise to schemas—that is, if $S \subseteq \mathcal{T}$ and $p \in \mathcal{P}$, we define $S(p) = \{t(p) \mid t \in S \wedge t(p) \neq \bot\}$. Note that a schema projection at a path is itself a schema. Next, we define what it means to transplant a subtree from one tree to another at a given path. Let $t \in \mathcal{T}$ and $p \in \mathcal{P}$ with $t(p) \in \mathcal{T}$. The *update* of t at p with t', written $t[p \mapsto t']$, is defined inductively on p as:

$$t[\bullet \mapsto t'] = t'$$
$$t[n/p \mapsto t'] = \left\{\begin{matrix} n \mapsto t(n)[p \mapsto t'] \\ m \mapsto t(m) \mid m \in \mathsf{dom}(t) \setminus \{n\} \end{matrix}\right\}$$

A schema S is *path consistent* if, whenever t and t' are in S, and for every path p, the result of updating t along p with $t'(p)$ is also in the schema. Formally, a schema S is path consistent iff, for all $t, t' \in S$ and $p \in \mathcal{P}$, it is the case that $t(p) \neq \bot$ and $t'(p) \neq \bot$ together imply $t[p \mapsto t'(p)] \in S$. For example, the schema $\{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{c}\}\}$ is trivially path consistent, as are all schemas whose members are "flat" trees.

Given a tree t and a path-consistent schema S, testing whether t belongs to S only requires a local check at every path. Formally, let the *domain set* of S, written $\mathsf{doms}(S)$, be the set of all domains of trees in S—i.e., $\mathsf{doms}(S) = \{\mathsf{dom}(t) \mid t \in S\}$. Then t belongs to S iff $\mathsf{dom}(t)$ belongs to $\mathsf{doms}(S)$ and, for every name $n \in \mathsf{dom}(t)$, $t(n)$ belongs to $S(n)$.

To complete the discussion of the role of schemas in synchronization, we must consider one final complication: for some inputs, there are *no* maximal runs belonging to the schema. Consider a run of the synchronizer on inputs $o = \{\mathtt{v}\}$, $a = \{\mathtt{w}, \mathtt{y}, \mathtt{z}\}$, and $b = \{\mathtt{w}, \mathtt{x}\}$, with respect to the schema $\{\{\mathtt{v}\}, \{\mathtt{w}, \mathtt{x}\}, \{\mathtt{w}, \mathtt{x}, \mathtt{y}\}, \{\mathtt{w}, \mathtt{x}, \mathtt{z}\}, \{\mathtt{w}, \mathtt{y}, \mathtt{z}\}\}$. For the a replica, the only tree that both belongs to the schema and preserves the additions and deletions relative to o is a itself. However, on the b side, there are three safe results that belong to the schema: $\{\mathtt{w}, \mathtt{x}\}$, $\{\mathtt{w}, \mathtt{x}, \mathtt{y}\}$, and $\{\mathtt{w}, \mathtt{x}, \mathtt{z}\}$. Notice that, since $\{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$ does not belong to the schema, we cannot include both $\mathtt{y}$ and $\mathtt{z}$ in b' without

backing out the addition of `x`. For each of these choices for b' there is a path p such that $b'(p) \neq a'(p)$, but a different choice of b' makes $b'(p) = a'(p)$. To rule out situations like this, we restrict the definition of the set of safe results so that whenever propagating all of the (otherwise non-conflicting) additions and deletions of immediate children yields an ill-formed result, a conflict occurs. This condition guarantees the existence of a maximal result because in such problematic cases a conflict occurs, and safety *requires* that the original replicas be returned when they are in conflict. Hence, the only safe run is trivially maximal. These new conflicts are called *schema domain conflicts* (see Definition 4 for a precise definition) because they can be detected using the domain set of the schema. Returning to the example, a schema domain conflict occurs at the root, since `y` and `z` cannot be validly added to b'.

Note, that the approach presented in this section is not the only way to ensure maximality in a schema-directed algorithm; we have considered several alternatives. First, we could do away with schema domain conflicts and require instead that schemas be closed under the "shuffling" of their domains with the domains of other trees in the schema. This approach amounts to declaring, by fiat, that the maximal result of every possible synchronization is present in the schema. For example, the schema above would need to additionally include $\{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$. We have not pursued this idea because it does not appear that shuffled schemas would be able to express the kinds of invariants needed by applications. Second, we could recognize schema domain conflicts, but, instead of requiring that the replicas remain unchanged, only require that the *domains* be unchanged. This approach would allow "deep" synchronization of subtrees, which has some obvious advantages. However, finding a natural notion of maximality for this variant has proved difficult. For these reasons, our first—simplest—proposal seems best.

5 Algorithm

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

Formally, the algorithm takes as inputs a path-consistent schema S, an archive o, and two current replicas a and b; it outputs a new archive o' and two new

$sync(S, o, a, b) =$

$\quad$ if $a = b$ then(a, a, b) $\quad$ – *equal replicas: done*

$\quad$ else if $a = o$ then (b, b, b) $\quad$ – *no change to a: propagate b*

$\quad$ else if $b = o$ then (a, a, a) $\quad$ – *no change to b: propagate a*

$\quad$ else if $o = \mathcal{X}$ then (o, a, b) $\quad$ – *unresolved conflict*

$\quad$ else if $a = \bot$ and $b \sqsubset o$ then (a, a, a) $\quad$ – *a deleted more than b*

$\quad$ else if $a = \bot$ and $b \not\sqsubset o$ then $(\mathcal{X}, a, b)$ $\quad$ – *delete/create conflict*

$\quad$ else if $b = \bot$ and $a \sqsubset o$ then (b, b, b) $\quad$ – *b deleted more than a*

$\quad$ else if $b = \bot$ and $a \not\sqsubset o$ then $(\mathcal{X}, a, b)$ $\quad$ – *delete/create conflict*

$\quad$ else $\quad$ – *proceed recursively*

$\quad\quad$ let $(o'(k), a'(k), b'(k)) = sync(S(k), o(k), a(k), b(k))$

$\quad\quad\quad$ $\forall k \in \mathsf{dom}(a) \cup \mathsf{dom}(b)$

$\quad\quad$ in

$\quad\quad\quad$ if $(\mathsf{dom}(a') \notin \mathsf{doms}(S))$ or $(\mathsf{dom}(b') \notin \mathsf{doms}(S))$

$\quad\quad\quad$ then $(\mathcal{X}, a, b)$ $\quad$ – *schema domain conflict*

$\quad\quad\quad$ else (o', a', b')

Fig. 2. Synchronization Algorithm

replicas a' and b'. We require that both a and b belong to $S_\bot$. The input archive may contain the conflict marker $\mathcal{X}$. As the replicas represent application data, they do not contain $\mathcal{X}$.

In the case where a and b are identical (i.e., the same tree or $\bot$), 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. If there was a conflict in the previous run, then it has not been resolved. If one replica is missing, then we check whether all the changes in the other replica are also deletions; formally, we check if the replica is included in the archive. If so, we consider the larger deletion (discarding the entire tree at this path) as subsuming the smaller; otherwise, we have a *delete/create conflict* and we return the original replicas.

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

(A naive implementation of this algorithm can perform many redundant equality checks in the common case when the replicas are almost equal—checking the topmost nodes for equality, failing, recursing on their immediate children,

checking these again for equality, etc. A combinatorial blowup in running time can be avoided by caching the results of equality checks.)

Appendix A contains a proof of the following correctness theorem:

Theorem 1 *Let $S \subseteq \mathcal{T}$ be a path-consistent schema. If $a, b \in S_\perp$ and the run* `sync`(S, o, a, b) *evaluates to o', a', b', then the run is* maximal.

6 Case Study: Address Books

We now present a brief case study, illustrating how schemas can be chosen so as to obtain desirable behavior from our generic synchronizer on structures of realistic complexity. The examples use an address book schema loosely based on the vCard standard [7], which embodies some of the tricky issues that can arise when synchronizing larger structures with varied substructure. We begin with a concrete notation for schemas over unordered trees, based on the Tree Logic of Dal Zilio et al [8].

Schemas Schemas are given by sets of mutually recursive equations of the form `X = S`, where `S` is an expression generated by the following grammar:

```
S ::= {} | n[S] | !(F)[S] | *(F)[S] | S,S | S|S | X
```

The symbols `n` and `F` range over names in $\mathcal{N}$ and finite sets in $\mathcal{P}(\mathcal{N})$, respectively. The first form of schema, `{}`, denotes the singleton set containing the empty tree; `n[S]` denotes the set of trees with a single child named `n`, where the subtree under `n` is in `S`; the wildcard schema `!(F)[S]` denotes the set of trees with *any* single child whose name is not in `F`, where the subtree under that child is in `S`; the other wildcard schema, `*(F)[S]`, denotes the set of trees with *any number of* children with names not in `F`, where the subtree under each child is in `S`. The set described by $\mathtt{S}_1\mathtt{|}\mathtt{S}_2$ is the union of the sets described by $\mathtt{S}_1$ and $\mathtt{S}_2$, while $\mathtt{S}_1\mathtt{,}\mathtt{S}_2$ denotes the set of trees of the form $t_1 \cdot t_2$ where t_1 belongs to $\mathtt{S}_1$ and t_2 to $\mathtt{S}_2$. Note that, as trees are unordered, the "`,`" operator is commutative (e.g., `n[X],m[Y]` and `m[Y],n[X]` are equivalent). We abbreviate `n[S]|{}` as `n?[S]`, and likewise `!(∅)[S]` as `![S]` and `*(∅)[S]` as `*[S]`. Variables like `X` are used to express recursive schemas. As usual, recursive variables may only appear contractively–e.g., we rule out definitions like `X = X` and `X = {}|X`.

In the following, all the schemas we write will be path consistent, a fact that can be verified syntactically: if a name appears twice in a node, like `m` in `m[X],n[Y]|m[X],o[Z]`, the subschemas associated with each occurrence of the name are textually identical.

Note that our schema formalism, like our generic synchronization algorithm,

```
  C = name[N],work[V],home?[V],org[O],email[E]
    | name[N],work?[V],home[V],org?[O],email[E]
  E = !(pref, alts)[{}] | pref[V],alts[VS]
  N = first[V],other?[VL],last[V]
  O = name[V],unit[V]
  V = ![{}]
 VL = head[V],tail[VL] | nil[{}]
 VS = *[{}]
```

Fig. 3. Address book schemas

does not distinguish between "structure" edges and "data" edges in trees—this distinction is just a matter of convention in each particular application. This choice makes our technical work easier by simplifying the formal objects (trees and schemas) that we manipulate.

Address Book Schema Here is a typical contact (the notation $[t_1; \ldots; t_n]$, which represents a list encoded as a tree, is explained below):

$$o = \left\{ \begin{array}{l} \texttt{name} \mapsto \{\texttt{first} \mapsto \texttt{Meg},\ \texttt{other} \mapsto [\texttt{Liz};\ \texttt{Jo}],\ \texttt{last} \mapsto \texttt{Smith}\}, \\ \texttt{email} \mapsto \{\texttt{pref} \mapsto \texttt{ms@c.edu},\ \texttt{alts} \mapsto \texttt{meg@s.com}\}, \\ \texttt{org} \mapsto \{\texttt{name} \mapsto \texttt{City U},\ \texttt{unit} \mapsto \texttt{CS Dept}\}, \\ \texttt{home} \mapsto \texttt{555-6666}, \\ \texttt{work} \mapsto \texttt{555-7777} \end{array} \right\}$$

There are two sorts of contacts—"professional" contacts, which contain mandatory work phone and organization entries, plus, optionally, a home phone; and "personal" ones, which have a mandatory home phone and, optionally, a work phone and organization information. Some contacts, like the one for `Meg`, belong to both sorts. Each contact also has fields representing name and email data.

The schema `C`, displayed in Figure 3, describes the record-like structure of both sorts of contacts. One level down, the subtrees that may validly appear below the `home` and `work` children are simple string values—i.e., trees with a single child leading to the empty tree—that belong to the `V` schema. The subtree below the `name` child in a valid contact belongs to the `N` schema, which describes trees with a record-like structure containing mandatory `first` and `last` children and optionally a child `other`. The `first` and `last` fields lead to values belonging to the `V` schema. The `other` field leads to a *list* of alternate names (e.g., middle names and nicknames) stored, for the sake of the example, in some particular order. Because our actual trees are unordered, we use a standard "cons cell" representation to encode ordered lists: $[t_1; \ldots; t_n]$ is encoded as $\{\texttt{head} \mapsto \texttt{t}_1, \texttt{tail} \mapsto \{\ldots \mapsto \{\texttt{head} \mapsto \texttt{t}_n, \texttt{tail} \mapsto \texttt{nil}\} \ldots\}\}$. The schema `VL` describes lists of values encoded like this. The email data for a contact is either a value or else a set of addresses with one distinguished "preferred" address. The `E` schema describes these structures using a union of a wildcard to

represent single values (excluding `pref` and `alts` to ensure path consistency) and a record-like structure with fields `pref` and `alts` to represent sets of addresses. The schema `VS` describes bushes with any number of children, each leading to the empty tree, which are a natural way of encoding sets of values. Finally, organization data is represented by trees with two children, `name` and `unit`, each leading to values, as described by the `O` schema.

The Need For Schemas To illustrate how and where schema conflicts can occur, let us see what can go wrong when *no* schema information is used. We consider four runs of the synchronizer using the universal schema `Any = *[Any]` (whose denotation is $\mathcal{T}$); each run shows a different example where schema-ignorant synchronization produces mangled results. In each case, the archive, o, is the tree above.

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

$$a = \left\{\begin{array}{l}\texttt{name}\mapsto\{\texttt{first}\mapsto\texttt{Meg},\ \texttt{other}\mapsto[\texttt{Liz};\ \texttt{Jo}],\ \texttt{last}\mapsto\texttt{Smith}\},\\ \texttt{email}\mapsto\{\texttt{pref}\mapsto\texttt{ms@c.edu},\ \texttt{alts}\mapsto\texttt{meg@s.com}\},\\ \texttt{home}\mapsto\texttt{555-6666}\end{array}\right\}$$

$$b = \left\{\begin{array}{l}\texttt{name}\mapsto\{\texttt{first}\mapsto\texttt{Meg},\ \texttt{other}\mapsto[\texttt{Liz};\ \texttt{Jo}],\ \texttt{last}\mapsto\texttt{Smith}\},\\ \texttt{email}\mapsto\{\texttt{pref}\mapsto\texttt{ms@c.edu},\ \texttt{alts}\mapsto\texttt{meg@s.com}\},\\ \texttt{org}\mapsto\{\texttt{name}\mapsto\texttt{City U},\ \texttt{unit}\mapsto\texttt{CS Dept}\},\\ \texttt{work}\mapsto\texttt{555-7777}\end{array}\right\}$$

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

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

Now suppose that the replicas are obtained by updating the trees along the path `name/first`, replacing `Meg` with `Maggie` in a and `Megan` in b (from now on, for the sake of brevity we only show the parts of the tree that are different from o and elide the rest): $o(\texttt{name/first}) = \texttt{Meg}$, $a(\texttt{name/first}) = \texttt{Maggie}$, and $b(\texttt{name/first}) = \texttt{Megan}$. Synchronizing with respect to the `Any` schema yields results where *both* names appear under `first`: $a'(\texttt{name/first}) = b'(\texttt{name/first}) = \{\texttt{Maggie}, \texttt{Megan}\}$. These results are ill-formed because they do not belong to the `V` schema, which describes trees that have a *single* child.

Next, consider updates to the email information where the a replica replaces the set of addresses in o with a single address, and b updates both `pref` and `alts` children in b: $o(\texttt{email}) = \{\texttt{pref} \mapsto \texttt{ms@c.edu},\ \texttt{alts} \mapsto \texttt{meg@s.com}\}$, $a(\texttt{email}) = \{\texttt{meg@s.com}\}$, and $b(\texttt{email}) = \{\texttt{pref} \mapsto \texttt{meg.smith@cs.c.edu},\ \texttt{alts} \mapsto \texttt{ms@c.edu}\}$. Synchronizing these trees with re-

spect to `Any` propagates the addition of the edge labeled `meg@s.com` from a to b' and yields conflicts on both `pref` and `alts` children, since both have been deleted in a but modified in b. The results after synchronizing are thus: $a'(\texttt{email}) = \texttt{meg@s.com}$ and $b'(\texttt{email}) = \{\texttt{meg@s.com},\ \texttt{pref} \mapsto \texttt{meg.smith@cs.c.edu},\ \texttt{alts} \mapsto \texttt{ms@c.edu}\}$. The second result, b', is ill-formed because it contains three children, whereas all the trees in the email schema `E` have either one or two children.

Next, consider changes to the list of names along the path `name/other`. Suppose that a removes both `Liz` and `Jo`, but b only removes `Jo`: $o(\texttt{name/other}) = [\texttt{Liz};\ \texttt{Jo}]$, $a(\texttt{name/other}) = []$, and $b(\texttt{name/other}) = [\texttt{Liz}]$. Comparing the a replica to o, both `head` and `tail` are deleted and `nil` is newly added. Examining the b replica, the tree under `head` is identical to the corresponding tree in o but deleted from a. The tree under `tail` is different from o but deleted from a. Collecting all of these changes, the algorithm yields these results: $a'(\texttt{name/other}) = \texttt{nil}$ and $b'(\texttt{name/other}) = \{\texttt{tail} \mapsto \texttt{nil},\ \texttt{nil}\}$. Here again, the second result, b', is ill-formed: it has children `tail` and `nil`, which is not a valid encoding of a list.

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

Next, we further explore the strengths (and weaknesses) of our algorithm by studying its behavior on the structures used in address books.

Values The simplest structures in our address books, string values, are represented as trees with a single child that leads to the empty tree and described by `![{}]`. When we synchronize two non-missing trees using this schema, there are three possible cases: (1) if either of a or b is identical to o then the algorithm set the results equal to the other replica; (2) if a and b are identical to each other but different from o then the algorithm preserves the equality; (3) if a and b are both different from o and each other then the algorithm reaches a

schema domain conflict and sets $o' = \mathcal{X}$, $a' = a$ and $b' = b$. These behaviors follow from the local alignment mechanism employed in the algorithm—children with the same name are identified across replicas and so identical values are aligned with each other and distinct values synchronized separately. In the first two scenarios, the differences in a and b (with respect to o) can be assembled into a value; in the third scenario, they cannot. That is, the algorithm enforces *atomic* updates to values, propagating updates from one side to the other only if o and either a or b are identical and otherwise leaving the replicas unchanged and signaling a schema domain conflict.

Sets Sets can be represented as bushes—nodes with many children, each labeled with the key of an element in the set. For example, sets of values belong to the schema `*[{}]`. When synchronizing two sets of values, the synchronization algorithm *never* reaches a schema conflict; it always produces a valid result, combining the additions and deletions of values from a and b. For example, given these three trees representing value sets: $o = \{$`meg@s.com`$\}$, with $a = \{$`ms@c.edu`, `meg.smith@cs.c.edu`$\}$ and $b = \{$`meg@s.com`, `meg.smith@cs.c.edu`$\}$ The synchronizer propagates the deletion of `meg@s.com` and the addition of two new children, `ms@c.edu` and `meg.smith@cs.c.edu`, yielding $a' = b' = \{$`ms@c.edu`, `meg.smith@cs.c.edu`$\}$, as expected.

Records Two sorts of record structures appear in the address book schema. The simplest records, like the one for organization data (`name[V],unit[V]`), have a fixed set of mandatory fields. Given two trees representing such records, the synchronizer aligns the common fields, which are *all* guaranteed to be present, and synchronizes the nested data one level down. It never reaches a schema domain conflict at the root of a tree representing such a record. Other records, which we call *sparse*, allow some variation in the names of their immediate children. For example, the contact schema uses a sparse record to represent the structure of each entry; some fields, like `org`, may be mandatory or optional (depending on the presence of other fields). As we saw in the preceding section, on some inputs—namely, when the updates to the replicas cannot be combined into a tree satisfying the constraint expressed by the sparse record schema—the synchronizer yields a schema conflict but preserves the sparse record structure.

Conclusion The examples in this section demonstrate that schemas are a useful addition to a synchronization algorithm: (1) we are guaranteed valid results in situations where a schema-blind algorithm would yield mangled results; (2) by selecting an appropriate encoding and schema for application data (moving keys high in the tree, etc.), we can tune the behavior of the generic algorithm to work well with many forms of data—both rigidly structured data (e.g., values and records) and unstructured data (e.g., sets of values). However, the local alignment strategy reaches its limits when confronted with ordered

and semi-structured data such as lists and documents, as we shall see next.

7 Synchronizing Lists

The address book schema in the previous section uses lists of values to represent the ordered collection of optional `other` names for a contact. We included this in the example to show how our synchronization algorithm behaves when applied to structures that are a little beyond its intended scope—i.e., mostly consisting of unordered or rigidly ordered data, but also containing small amounts of list-structured data. Lists present special challenges, because we would like the algorithm to detect updates both to elements and to their relative position. But our local alignment strategy matches up list elements by *absolute* position, leading to surprising results on some inputs. In this section we illustrate the problem and propose an extension of the algorithm that provides a better treatment of lists.

To begin with, it is worth noting that, in many cases, updates to lists can be propagated successfully even by the algorithm we have given. If either replica is identical to the archive, or if each replica modifies a disjoint subset of the elements of the list (leaving the list spine intact), then the synchronizer merges the changes successfully. At each step in a recursive tree walk only one of the elements will have changed, as in the following example (writing out the low-level tree representation for each tree):

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

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

$$
a' = b' = \{\texttt{head}\mapsto\texttt{Elizabeth},\ \texttt{tail}\mapsto\{\texttt{head}\mapsto\texttt{Joanna},\ \texttt{tail}\mapsto\texttt{nil}\}\}
$$

There are some inputs, however, where synchronizing lists using the local alignment strategy and simple cons-cell encoding produces strange results. Consider a run on the following inputs: $o = [\texttt{Liz};\ \texttt{Jo}]$, $a = [\texttt{Jo}]$ and $b = [\texttt{Liz};\ \texttt{Joanna}]$. Considering the changes that were made to each list from a high-level—a removed the head and b renamed the second element—the result calculated for b' is surprising: $[\texttt{Jo};\ \texttt{Joanna}]$. The algorithm does not recognize that `Jo` and `Joanna` should be aligned (because `Joanna` was obtained by renaming `Jo`). Instead, it aligns pieces of the list by absolute position, matching `Jo` with `Liz` and `nil` with $[\texttt{Joanna}]$.

$sync(S, o, a, b) =$
  if $a = b$ then(a, a, b) – *equal replicas: done*
    ...
  else if $S = List(T)$ for some T, then $diff3(T, o, a, b)$ – *special case for lists*
  else – *proceed recursively*
    let $(o'(k), a'(k), b'(k)) = sync(S(k), o(k), a(k), b(k))$
      $\forall k \in \mathsf{dom}(a) \cup \mathsf{dom}(b)$
    in ...

Fig. 4. Synchronization With Lists

Of course, it is not very surprising that our purely local algorithm does not do a good job with lists. Detecting changes in relative position in a list requires a global *comparison* of its current and previous states, and our algorithm makes all its alignment decisions looking at just one structure at a time. We are experimenting with a number of possible avenues for extending the local algorithm to deal better with lists. A full treatment of this issue goes beyond the scope of this paper—in particular, it is not clear how to reconcile powerful tree-differencing techniques (e.g., [3–5]) with our core goal of maintaining well-formedness of synchronized structures, or with the mostly unordered nature of many of the trees we work with. However, we have implemented a very simple refinement to the algorithm, which preserves its fundamentally local character while handling many more cases of updates to lists.

The refined algorithm, sketched in Figure 4, uses the synchronization schema S to test whether the replicas are actually representations of lists. If they are, and if none of the trivial cases apply, then the algorithm invokes a different procedure, called *diff3*, to analyze how the elements of the list should be aligned; otherwise, it continues with the same recursive case as in the simple algorithm. The details of what happens inside *diff3* are not critical—any list synchronization algorithm will do; in the version we have implemented, *diff3* performs a global alignment of the list structures at the top of o, a, and b by calculating maximal common sub-sequences between o and a and between o and b to decide where changes have occurred, just like the standard UNIX text utility `diff3`. It then propagates non-conflicting change regions from a to b and vice versa. On regions where changes conflict, it calls back to *sync* recursively on corresponding triples of elements from o, a, and b, passing T, the type of the list elements, as the synchronization schema; if a conflict region does not have the same number of elements in o, a, and b, then a conflict is signalled and the regions are left as-is in the output.

The result of this refinement is that unordered and rigidly ordered parts of the input structures are synchronized precisely as before, while embedded lists of atomic values are synchronized just as the `diff3` utility would do. Embedded lists of more structured data are synchronized first as with `diff3` and then

by recursively synchronizing corresponding subtrees. The well-formedness of the resulting structures is guaranteed by construction. Of course, the cost of this extra functionality is that the clear and simple specification of the synchronizer's behavior (Appendix A) is no longer applicable. In particular, the notions of *change, conflict*, and *maximality* all become much more subtle.

8 Related Work

Harmony combines a synchronization component with a view update component for dealing with alignment and heterogeneous replicas. The latter component and related work on the view update problem are described elsewhere [1]. Here, we focus on work related to optimistic replication and synchronization. For further information on this area, we recommend an excellent article by Saito and Shapiro [9] surveying the area of optimistic replication. In the taxonomy of the survey, Harmony is a multi-master state-transfer system, recognizing sub-objects and manually resolving conflicts. Harmony is further characterized by some distinctions not covered in that survey: it is generic, loosely coupled to applications, and able to synchronize heterogeneous representations, and it supports unsupervised operation, where Harmony does as much work as it can, leaving conflicts for later resolution. This last property, in particular, demands that the synchronizer's behavior is intuitive and easy to predict.

Synchronization Architectures Harmony's state-based architecture only requires loose coupling between the synchronizer and applications. By contrast, in an operation-based approach, applications must be capable of providing the synchronizer with a log of the operations performed on each replica. The behavior of the synchronizer, then, is to merge logs so that, after quiescence, when each replica applies its merged log to the last synchronized state, all the replicas share a uniform state. There are, broadly speaking, three alternatives when merging logs: (1) reorder operations on all replicas to achieve an identical schedule (cf. Bayou [10]), (2) partially reorder operations, exploiting semantic knowledge to leave equivalent sequences unordered (cf. IceCube [11]) or (3) perform no reordering, but transform the operations themselves, so that the different schedules on each different replica all have a uniform result (cf. Molli et al [12]). Although we favor the state-based approach, because it facilitates loose coupling, there are some advantages to working with operations. State-based architectures have less information available at synchronization time; they cannot exploit the knowledge of temporal sequencing available in operation logs. When operation logs are available, systems can sometimes determine that two modifications are not in conflict if one is in the operation history of the other. Further, in an operation-based system, the designer

can often choose operations that encode high-level application semantics. The synchronizer then manipulates operations that are close to the actual user operations. Treating user-level operations as primitives makes it likely from the perspective of the user that, even under conflict, the state computed by the synchronizer will be well-formed.

Other state-based synchronizers include FCDP [5] and most file system synchronizers. File system synchronizers (such as [13–18]) and PDA synchronizers (such as Palm's HotSync), are generally state-based but not generic. An interesting exception is DARCS [19], a hybrid state-/operation-based revision control system built on a "theory of patches." FCDP is a generic, state-based reconciler parameterized by ad-hoc translations from heterogeneous concrete representations to XML and back again. Broadly speaking, FCDP can be considered an instance of the Harmony architecture—but without the formal underpinnings. FCDP is less generic (our lens language is easier to extend to new applications), but it is better able to deal with certain edits to documents than Harmony. FCDP achieves this by fixing a specific semantics for ordered lists that is particularly suited for document editing.

On the other side, IceCube [11,20] is a generic, operation-based reconciler that is parameterized over an algebra of operations specific to the application data being synchronized and by a set of syntactic/static and semantic/dynamic ordering constraints on these operations. Molli et al [21,22,12], have also implemented a generic operation-based reconciler, using the technique of operational transformation. Their synchronizer is parameterized on transformation functions for all operations, which must obey certain conditions, and also obeys a formal specification.

It is worth noting that 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. Bengal [23] is an interesting example of the hybrid approach; it is a loosely coupled synchronizer that uses operation logs as an optimization to avoid scanning the entire replica during update detection. Bengal supports updates to a wide range of databases via OLE/COM hooks, but is not heterogeneous because reconciliation may only occur between replicas of the same database.

Conflicts and Convergence Harmony, unlike many reconcilers, does not guarantee convergence in the case of conflicts. When conflicts occur, reconcilers

can choose one of three broad strategies.

- They can settle all conflicts by fiat, for example by choosing the variant with the latest modification time and discarding the other.
- They can converge without resolving the conflict. In other words, they can keep enough information to record *both* conflicting updates, and converge to a single state containing 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.

Most modern reconcilers will not simply discard updates (they obey a "no lost updates" policy). Harmony chooses the third option (divergence) over the second (unconditional convergence).

Systems such as Ficus [24], Rumor [15], Clique [16], Bengal [23], and TAL/S5 [21,22,12] converge by making additional copies of primitive objects that conflict and renaming one of the copies. Additionally, conflicts in Bengal may be handled by user-programmed *conflict resolvers*. CVS embeds markers in the bodies of files where conflicts occurred. By contrast, systems such as Harmony and IceCube [11] do 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 Appendix A we show that Harmony is a *maximal synchronizer*, propagating as many safe changes as possible.

Like Harmony, the synchronizer of Molli et al [21,22,12] uses formal specifications to ensure safety, but unlike Harmony it chooses convergence over persistence of user changes. As a result, Molli's synchronizer is satisfied with recording multiple conflicting versions in the reconciled replicas, and its specification is limited to describing the correctness of its transformation functions.

Operational transforms resolve conflicting schedules by transforming local operations to undo the local operation, performing the remote operation, and finally redoing the local operation. Understanding the correct behavior of "undo" in a collaborative environment is an additional prerequisite to the correct behavior of operational transformation. Munson and Dewan [25] note that group "undo" may remove the need for a merge capability in optimistic replication. Prakash and Knister [26] provide formal properties that individual primitive operations in a system must satisfy in order to be undoable in a groupware setting. Abowd and Dix [27] 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.

Heterogeneity Answering queries from heterogeneous data sources is a well-studied area in the context of data integration [28–31]. If we consider the problem of augmenting a data integration system with view update (another well-studied area—see [1] for a survey), then the result can be used to implement an optimistic replication system that reconciles conflicts between heterogeneous data sources However, to the best of our knowledge, no other generic synchronizers support reconciliation over truly heterogeneous replicas. FCDP [5] is designed to be generic, but the genericity is limited to using XML as the internal representation, and it 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 (cf. [17,16]). Others map complex file attributes (e.g. Apple resource forks) into directories, rather than files, on the remote replicas.

Alignment 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 (cf. [32–36]). 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. For example, the main difference between Unison [17,37] (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, developing the line of inquiry sketched in Section 7. Of particular interest are `diff3`'s XML based descendants: Lindholm's 3DM [3], the work of Chawathe et al [4], and FCDP [5].

9 Conclusions and Future Work

We have described the design of Harmony, a generic synchronization framework for tree-structured data based around the idea of *schema-directed synchronization*. A prototype implementation is available for download from the

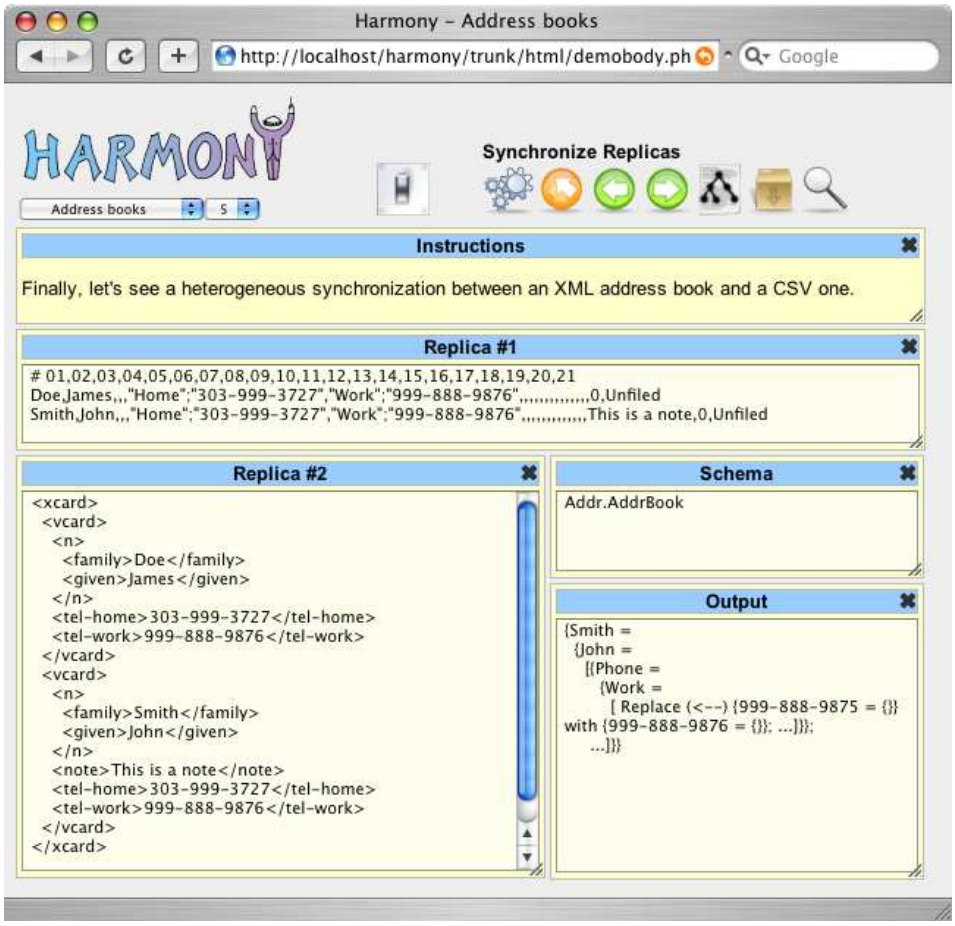


Fig. 5. Web demo screenshot

Harmony web site, `http://www.seas.upenn.edu/~harmony`. The web site also includes a live demo that allows experimenting with synchronization of small tree structures; Figure 5 shows a screenshot of the demo, operating on heterogeneous address book formats. The prototype has been used to construct specific synchronizers for a number of data formats, including several calendar formats (Palm Datebook, Unix ical, and iCalendar), bookmarks (Mozilla, Safari, and Internet Explorer), address books, application preference files, drawings, and bibliographic databases. Some of these instances are now in daily use.

One ongoing project is combining the core features of Harmony with a more sophisticated treatment of ordered structures, continuing along the lines sketched in Section 7. More speculatively, although the Harmony framework has been designed with tree synchronization in mind, it may be generalizable to richer structures such as DAGs. Along the same lines, another ongoing effort in our group aims to apply the ideas in Harmony (both lenses and synchronization) to the domain of relational data [38]. We are also interested in studying schema-aware data synchronization using classical tree processing formalisms, including tree automata and transducers.

Acknowledgements

The Harmony project was begun in collaboration with Zhe Yang; Zhe contributed numerous insights whose genetic material can be found (generally in much-recombined form) in this paper. Trevor Jim provided the initial push to start the project by observing that the next step beyond the Unison file synchronizer (of which Trevor was a co-designer) would be synchronizing XML. Jonathan Moore, Owen Gunden, Malo Denielou, and Stéphane Lescuyer have collaborated with us on many aspects of Harmony's design and implementation. Ongoing work with Sanjeev Khanna and Keshav Kunal on extensions of Harmony's synchronization algorithm has deepened our understanding of the core mechanisms presented here. Conversations with Martin Hofmann, Zack Ives, Nitin Khandelwal, William Lovas, Kate Moore, Cyrus Najmabadi, Stephen Tse, Steve Zdancewic, and comments from anonymous reviewers helped sharpen our ideas. Harmony is supported by the National Science Foundation under grants ITR-0113226, *Principles and Practice of Synchronization* and IIS-0534592 *Linguistic Foundations for XML View Update*. Nathan Foster is also supported by an NSF GRF.

References

[1] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, Combinators for bi-directional tree transformations: A linguistic approach to the view update problem, in: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California, 2005, pp. 233–246.

[2] M. B. Greenwald, S. Khanna, K. Kunal, B. C. Pierce, A. Schmitt, Agreeing to agree: Conflict resolution for optimistically replicated data, in: S. Dolev (Ed.), International Symposium on Distributed Computing (DISC), 2006.

[3] T. Lindholm, XML three-way merge as a reconciliation engine for mobile data, in: ACM Workshop on Data Engineering for Wireless and Mobile Access (MobiDE), San Diego, California, 2003, pp. 93–97.

[4] S. S. Chawathe, A. Rajamaran, H. Garcia-Molina, J. Widom, Change detection in hierarchically structured information, ACM SIGMOD Record 25 (2) (1996) 493–504.

[5] M. Lanham, A. Kang, J. Hammer, A. Helal, J. Wilson, Format-independent change detection and propoagation in support of mobile computing, in: Brazilian Symposium on Databases (SBBD), Gramado, Brazil, 2002, pp. 27–41.

[6] F. Bancilhon, N. Spyratos, Update semantics of relational views, ACM Transactions on Database Systems 6 (4) (1981) 557–575.

[7] T. Howes, M. Smith, F. Dawson, RFC 2425: A MIME content-type for directory information (Sep. 1998).

[8] S. Dal Zilio, D. Lugiez, C. Meyssonnier, A Logic You Can Count On, in: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Venice, Italy, ACM Press, 2004, pp. 135–146.

[9] Y. Saito, M. Shapiro, Optimistic replication, Computing Surveys 37 (1) (2005) 42–81.

[10] W. K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, Designing and implementing asynchronous collaborative applications with Bayou, in: ACM Symposium on User Interface Software and Technology (UIST), Banff, Alberta, 1997, pp. 119–128.

[11] A.-M. Kermarrec, A. Rowstron, M. Shapiro, P. Druschel, The IceCube approach to the reconciliation of diverging replicas, in: ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Newport, Rhode Island, 2001, pp. 210–218.

[12] P. Molli, G. Oster, H. Skaf-Molli, A. Imine, Using the transformational approach to build a safe and generic data synchronizer, in: ACM SIGGROUP Conference on Supporting Group Work (GROUP), Sanibel Island, Florida, 2003, pp. 212–220.

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

[14] T. W. P. Jr., R. G. Guy, J. S. Heidemann, D. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, G. J. Popek, Perspectives on optimistically replicated, peer-to-peer filing., Softw., Pract. Exper. 28 (2) (1998) 155–180.

[15] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, G. J. Popek, Rumor: Mobile data access through optimistic peer-to-peer replication, in: Proceedings of the ER Workshop on Mobile Data Access, 1998, pp. 254–265.

[16] B. Richard, D. M. Nioclais, D. Chalon, Clique: a transparent, peer-to-peer collaborative file sharing system, in: International Conference on Mobile Data Management (MDM), Melbourne, Australia, 2003.

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

[18] N. Ramsey, E. Csirmaz, An algebraic approach to file synchronization, in: Proceedings of the 8th European Software Engineering Conference, ACM Press, 2001, pp. 175–185.

[19] D. Roundy, The DARCS system, `http://abridgegame.org/darcs/` (2004).

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

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

[22] A. Imine, P. Molli, G. Oster, M. Rusinowitch, Proving correctness of transformation functions in real-time groupware, in: ACM Conference on Computer Supported Cooperative Work (CSCW), Helsinki, Finland, 2003.

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

[24] P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, G. J. Popek, Resolving file conflicts in the ficus file system, in: USENIX Summer Conference Proceedings, 1994, pp. 183–195.

[25] J. P. Munson, P. Dewan, A flexible object merging framework, in: ACM Conference on Computer Supported Cooperative Work (CSCW), Chapel Hill, North Carolina, 1994, pp. 231–242.

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

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

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

[29] S. Abiteboul, Querying semi-structured data, in: International Conference on Database Theory (ICDT), Delphi, Greece, 1997, pp. 1–18.

[30] A. Y. Halevy, Theory of answering queries using views, ACM SIGMOD Record 29 (4) (2000) 40–47.

[31] I. Tatarinov, Z. G. Ives, A. Y. Halevy, D. S. Weld, Updating XML, in: ACM SIGMOD Symposium on Management of Data (SIGMOD), Santa Barbara, California, 2001.

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

[33] T. Milo, S. Zohar, Using schema matching to simplify heterogeneous data translation, in: International Conference on Very Large Data Bases (VLDB), New York, New York, 1998.

[34] C. Beeri, T. Milo, Schemas for integration and translation of structured and semi-structured data, in: International Conference on Database Theory (ICDT), Jerusalem, Israel, 1999.

[35] A. Doan, P. Domingos, A. Y. Halevy, Reconciling schemas of disparate data sources: A machine-learning approach, in: ACM SIGMOD Symposium on Management of Data (SIGMOD), Santa Barbara, California, 2001.

[36] J. Madhavan, P. A. Bernstein, E. Rahm, Generic schema matching with Cupid, in: International Conference on Very Large Data Bases (VLDB), Roma, Italy, 2001, pp. 49–58.

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

[38] A. Bohannon, J. A. Vaughan, B. C. Pierce, Relational lenses: A language for updateable views, in: Principles of Database Systems (PODS), 2006, extended version available as University of Pennsylvania technical report MS-CIS-05-27.

A Specification

This technical appendix contains a formal specification of the properties of the synchronization algorithm presented in Section 5. Our presentation follows the basic approach used for specifying the Unison file synchronizer [17,37]. We start with a few auxiliary definitions which are needed in the formal statements of safety and maximality. Both safety and maximality are based on a notion of *local similarity*, which relates two trees if their top-level nodes are similar—i.e., intuitively, if both are present, both are missing, or both represent conflicts.

Definition 2 (Local similarity) *Two elements of $\mathcal{T}_{\mathcal{X}\perp}$ are locally similar, written $t \sim t'$, iff (1) $t = t' = \mathcal{X}$ or (2) $t = t' = \perp$ or (3) $t \neq \perp \wedge t \neq \mathcal{X} \wedge t' \neq \perp \wedge t' \neq \mathcal{X}$.*

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

Definition 3 (Local conflict) *An archive o and replicas (a, b) have a local conflict, written $\mathsf{localconflict}(o, a, b)$, iff either (1) $(o = \mathcal{X}) \wedge (a \neq b)$ or (2) $(o \neq a) \wedge (o \neq b) \wedge (a \nsim b) \wedge (a = \perp \implies b \not\sqsubseteq o) \wedge (b = \perp \implies a \not\sqsubseteq o)$.*

Intuitively, replicas (a, b) have a local conflict if there is an unresolved conflict recorded in the archive o, or if they have both changed since the state recorded in the archive but are not locally similar and are in a delete/create conflict. Schema domain conflicts are defined via a local test on domains.

Definition 4 (Schema domain conflict) *Let S be a schema. An archive o and replicas (a, b) have a* schema domain conflict*, written $\mathsf{sdomconflict}(S, o, a, b)$, iff $(a \neq \perp) \wedge (b \neq \perp) \wedge (\mathsf{mdom}(o, a, b) \notin \mathsf{doms}(S) \vee \mathsf{mdom}(o, b, a) \notin \mathsf{doms}(S))$ where $\mathsf{mdom}(o, a, b) = \{k \in \mathsf{dom}(a) \mid a(k) \not\sqsubseteq o(k)\} \cup (\mathsf{dom}(b) \setminus \mathsf{dom}(o)) \cup (\mathsf{dom}(a) \cap \mathsf{dom}(b))$.*

The set of names $\mathsf{mdom}(o, a, b)$ is the maximal tree domain that results from propagating into replica a all the changes to immediate children from replica b, that also respects local conflicts. It contains every name under which there is a change that is not simply a deletion in replica a, all names added by replica b, and all the names preserved in both a and b.

Definition 5 (Conflict) *Let S be a schema. An archive o and replicas (a, b) have a* conflict*, written $\mathsf{conflict}(S, o, a, b)$, iff $\mathsf{localconflict}(o, a, b)$ or $\mathsf{sdomconflict}(S, o, a, b)$.*

Safety and maximality are both expressed as properties of *runs*.

Definition 6 (Synchronizer run) *A synchronizer* run *is a tuple* (S, o, a, b, o', a', b') *comprising a schema* S *together with trees representing the archive and replicas supplied as inputs,* (o, a, b)*, and computed as outputs,* (o', a', b')*, by the synchronizer.*

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

Definition 7 (Locally safe run) *A synchronizer run is* locally safe*, written* $\mathsf{locallysafe}(S, o, a, b, o', a', b')$*, iff:*

(1) It does not discard changes: $(o \nsim a \implies a' \sim a)$ *and* $(o \nsim b \implies b' \sim b)$*,*
(2) It does not "make up" content: $(a \nsim a' \implies b \sim a')$ *and* $(b \nsim b' \implies a \sim b')$ *and* $(o' \neq \mathcal{X} \implies o' \sim a' \wedge o' \sim b')$*,*
(3) It stops at conflicting paths (leaving replicas in their current states and recording the conflict): $\mathsf{conflict}(o, a, b) \implies (a', b', o') = (a, b, \mathcal{X})$*,*
(4) The results belong to the schema (or are missing): $a' \in S_\perp$ *and* $b' \in S_\perp$*.*

Definition 8 (Safe run) *A synchronizer run* (S, o, a, b, o', a', b') *is* safe*, written* $\mathsf{safe}(S, o, a, b, o', a', b')$ *iff it is locally safe on every path* p*:*

$$\forall p \in \mathcal{P}.\ \mathsf{locallysafe}(S(p), o(p), a(p), b(p), o'(p), a'(p), b'(p))$$

Definition 9 (Maximal run) *A synchronizer run* $(S,o,a,b,\ o',a',b')$ *is* maximal *iff it is safe and propagates at least the same changes as any other safe run, i.e.*

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

Next we give a precise specification of Harmony's synchronization algorithm.

Theorem 10 *Let* S *be a path-consistent schema. If* $a, b \in S_\perp$ *and* $\mathtt{sync}(S, o, a, b)$ *evaluates to* (o', a', b')*, then* (S, o, a, b, o', a', b') *is maximal.*

Before proving Theorem 10, we prove a few technical lemmas. First we note that path consistency is stable under projection on a single name.

Lemma 11 *Let* n *be a name and* S *a path-consistent schema; the schema* $S(n)$ *is also path consistent.*

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

$t'_n(p) \in \mathcal{T}$, we have

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

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

The key property enjoyed by path-consistent schemas is that we can assemble well-formed trees (belonging to the schema) out of well-formed subtrees.

Lemma 12 *Let S be a path-consistent schema. If $\mathsf{dom}(t) \in \mathsf{doms}(S)$ and for each $k \in \mathsf{dom}(t)$ we have $t(k) \in S(k)$ then $t \in S$.*

Proof. Let $\mathsf{dom}(t) = \{n_1, \ldots, n_j\}$. As $t(k) \in S(k)$ for every $k \in \mathsf{dom}(t)$, for each $k \in \{n_1 \ldots n_j\}$ there exists a tree $t'_k \in S$ with $t'_k(k) = t(k)$. Moreover, as $\mathsf{dom}(t) \in \mathsf{doms}(S)$, there is a tree $t'' \in S$ with $\mathsf{dom}(t'') = \mathsf{dom}(t)$. We now show by a finite induction on $i \leq j$ that $t''[n_1 \mapsto t'_{n_1}(n_1)] \ldots [n_i \mapsto t'_{n_i}(n_i)] \in S$.

Case $(i = 0)$**:** immediate as $t'' \in S$.
Case $(i = k + 1)$**:** By IH, $t''' = t''[n_1 \mapsto t'_{n_1}(n_1)] \ldots [n_k \mapsto t'_{n_k}(n_k)] \in S$. We also have $t'_{n_{k+1}} \in S$ by assumption. Moreover, both $t'''(n_{k+1}) = t''(n_{k+1}) \in \mathcal{T}$ and $t'_{n_{k+1}}(n_{k+1}) \in \mathcal{T}$. By path consistency for S, we hence have $t'''[n_{k+1} \mapsto t'_{n_{k+1}}(n_{k+1})] \in S$. We conclude that $t = t''[n_1 \mapsto t'_{n_1}(n_1)] \ldots [n_j \mapsto t'_{n_j}(n_j)] \in S$. $\square$

The next lemma records some examples of safe runs (both to illuminate the definitions and to shorten the proof of Theorem 10).

Lemma 13 *If $a \in S_\perp$ and $b \in S_\perp$ then the following synchronizer runs are safe:*

(1) $(S, o, a, b, \mathcal{X}, a, b)$
(2) (S, o, a, a, a, a, a)
(3) (S, o, o, b, b, b, b)
(4) (S, o, a, o, a, a, a)
(5) $(S, o, \perp, b, \perp, \perp, \perp)$, *assuming* $b \sqsubset o$
(6) $(S, o, a, \perp, \perp, \perp, \perp)$, *assuming* $a \sqsubset o$

Proof. Each case is straightforward.

Now we are ready to come back to Theorem 10.

Proof of Theorem 10. 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 $(a = b)$**:** $\texttt{sync}(S, o, a, b) = (a, a, b)$

By Lemma 13(2) the run (S, o, a, a, a, a, a) is safe. We must show that (S, o, a, a, a, a, a) is maximal. The first condition for maximality is immediate as for all paths p, we have $a'(p) \sim b'(p)$. The second condition is also satisfied, since $o' = a$, hence we have $o'(p) \neq \mathcal{X}$ for all paths p.

Case $(a = o)$**:** $\texttt{sync}(S, o, a, b) = (b, b, b)$

By Lemma 13(3) the run (S, o, o, b, b, b, b) is safe. We must show that (S, o, o, b, b, b, b) is maximal. The first condition for maximality is immediate as for all paths p, we have $a'(p) \sim b'(p)$. The second condition is also satisfied, since $o' = b$, hence we have $o'(p) \neq \mathcal{X}$ for all paths p.

Case $(b = o)$**:** $\texttt{sync}(S, o, a, b) = (a, a, a)$

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

Case $(o = \mathcal{X})$**:** $\texttt{sync}(S, o, a, b) = (\mathcal{X}, a, b)$

By Lemma 13(1) the run $(S, \mathcal{X}, a, b, \mathcal{X}, a, b)$ is safe. We must now show that the run is maximal. The predicate $\mathsf{localconflict}(\mathcal{X}, a, b)$ is satisfied because we know that $o = \mathcal{X}$ and $a \neq b$ (as the first case of the algorithm did not apply). Therefore, by safety condition (3), the only safe run is $(S, o, a, b, \mathcal{X}, a, b)$, hence it is maximal.

Case $(a = \bot)$**:** there are two subcases to consider.

Subcase $(b \sqsubset o)$**:** $\texttt{sync}(S, o, a, b) = (\bot, \bot, \bot)$

By Lemma 13(5) the run $(S, o, \bot, b, \bot, \bot, \bot)$ is safe if $b \sqsubset o$. The first maximality condition is immediate as for all paths p, we have $a'(p) \sim b'(p)$. The second condition is also satisfied since $o' = \bot$, hence we have $o'(p) \neq \mathcal{X}$ for all paths p.

Subcase $(b \not\sqsubset o)$**:** $\texttt{sync}(S, o, a, b) = (\mathcal{X}, \bot, b)$

By Lemma 13(1) the run $(S, o, \bot, b, \mathcal{X}, \bot, b)$ is safe. We must now prove that the run is maximal. None of the previous cases of the algorithm apply, so we must have $b \neq a = \bot$, $o \neq a = \bot$, and $b \neq o$. Since $a = \bot$ and $b \neq \bot$, we have $a \nsim b$. Moreover, we have $b \not\sqsubset o$ and $b \neq \bot$. Hence the predicate $\mathsf{localconflict}(o, a, b)$ is satisfied. As before, by safety condition (3), the only safe run is $(S, o, a, b, \mathcal{X}, \bot, b)$, hence it is maximal.

Case $(b = \bot)$**:** symmetric to the previous case, inverting the roles of a and b and using Lemma 13(6) instead of Lemma 13(5) in the first subcase.

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

By Lemma 11, each of the schemas $S(k)$ are path consistent for $k \in \mathsf{dom}(a) \cup \mathsf{dom}(b)$. By the definition of schema projection, for each k we have $a(k), b(k) \in S(k)_\bot$. Thus, by the IH (which applies as $a \neq \bot \neq b$, hence the sum of the depths decreases), each recursive sub-run is maximal. In particular, each sub-run is safe and gives results in $S(k)_\bot$.

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

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

Subcases $(\mathcal{X}, \bot, \bot)$**,** $(\bot, \bot, \bot)$**,** $(t_o, \bot, \bot)$**:** Cannot happen because the given child $k \in \mathsf{dom}(a) \cup \mathsf{dom}(b)$.

Subcases $(\bot, t_a, \bot)$**,**$(\bot, t_a, t_b)$**:** Since $o(k) \nsim a(k)$ we get by local safety condition (1) that $a'_r(k) \sim t_a$ and in particular $a'_r(k) \neq \bot$, hence $k \in \mathsf{dom}(a'_r)$. On the other hand, since $k \in \{n \in \mathsf{dom}(a) \mid a(n) \not\sqsubset o(n)\}$ we also have that $k \in \mathsf{mdom}(o, a, b)$.

Subcases $(\mathcal{X}, t_a, \bot)$**,**$(\mathcal{X}, t_a, t_b)$**:** Same as previous case, as $t_a \not\sqsubset \mathcal{X}$.

Subcases $(t_o, \bot, t_b)$**,**$(\mathcal{X}, \bot, t_b)$**:** Since $o(k) \nsim a(k)$ we get by local safety condition (1) that $a'_r(k) \sim \bot$ and therefore $a'_r(k) = \bot$, so $k \notin \mathsf{dom}(a'_r)$. On the other hand, since $a(k) = \bot$ we have that $k \notin \mathsf{dom}(a)$ so especially $k \notin \{n \in \mathsf{dom}(a) \mid a(n) \not\sqsubset o(n)\}$, $k \notin \mathsf{dom}(a) \cap \mathsf{dom}(b)$, and since $o(k) \neq \bot$ we have that $k \notin \{n \in \mathsf{dom}(b) \mid o(n) = \bot\}$, hence also $k \notin \mathsf{mdom}(o, a, b)$.

Subcase $(\bot, \bot, t_b)$**:** By Lemma 13(3) the run $(S(k), \bot, \bot, t_b, t_b, t_b, t_b)$ is safe. Letting a'' and b'' stand for the replica outputs of this run, which are both t_b, we have that $a'' \sim b''$. By the maximality of the recursive sub-run we get that $a'_r(k) \sim b'_r(k)$. As $t_b \nsim \bot$, we have $b'_r(k) \sim t_b$ thus $b'_r(k) \neq \bot$ by local safety (1), hence $a'_r(k) \neq \bot$. From this, we have $k \in \mathsf{dom}(a'_r)$. On the other hand, since $k \in (\mathsf{dom}(b) \setminus \mathsf{dom}(o))$ we also have that $k \in \mathsf{mdom}(o, a, b)$.

Subcase $(t_o, t_a, \bot)$**:** We first consider the case $t_a \not\sqsubset t_o$, which immediately implies $t_a \neq t_o$. In this case, as we have $t_a \neq t_o$, $t_o \neq \bot$, $t_a \nsim \bot$, $t_a \neq \bot$, and $t_a \not\sqsubset t_o$, the predicate $\mathsf{localconflict}(t_o, t_a, \bot)$ is satisfied. By local safety condition (3), we have $a'_r(k) = t_a$ so $k \in \mathsf{dom}(a'_r)$. On the other hand, since $a(k) \not\sqsubset o(k)$ we have that $k \in \{n \in \mathsf{dom}(a) \mid a(n) \not\sqsubset o(n)\}$, hence also $k \in \mathsf{mdom}(o, a, b)$.

We now consider the other case, $t_a \sqsubset t_o$. By Lemma 13(6) the run $(S(k), t_o, t_a, \bot, \bot, \bot, \bot)$ is safe as $t_a \sqsubset t_o$. Letting a'' and b'' stand for the replica outputs of this run, which are both $\bot$, we have that $a'' \sim b''$. Therefore, by maximality of the recursive sub-run we get that $a'_r(k) \sim b'_r(k)$. Next we show that $a'_r(k) = \bot$. Since $t_o \nsim \bot$ we have $b'_r(k) \sim \bot$ by local safety (1) and hence $a'_r(k) = \bot$. It follows that $k \notin \mathsf{dom}(a'_r)$. On the other hand, since $a(k) \sqsubset o(k)$ we have $k \notin \{n \in \mathsf{dom}(a) \mid a(n) \not\sqsubset o(n)\}$, and since $k \notin \mathsf{dom}(b)$ we have $k \notin \mathsf{dom}(a) \cap \mathsf{dom}(b)$ and $k \notin (\mathsf{dom}(b) \setminus \mathsf{dom}(o))$, hence also $k \notin \mathsf{mdom}(o, a, b)$.

Subcase (t_o, t_a, t_b)**:** We first show that $a'_r(k) \neq \bot$, by contradiction. Assume $a'_r(k) = \bot \nsim t_a$, hence by local safety condition (2) we have $t_b \sim a'_r(k) = \bot$, a contradiction. Thus $k \in \mathsf{dom}(a'_r)$. As $k \in \mathsf{dom}(a) \cap \mathsf{dom}(b)$, we conclude by $k \in \mathsf{mdom}(o, a, b)$.

By a symmetric argument, we can show that $\mathsf{dom}(b'_r) = \mathsf{mdom}(o, b, a)$. Returning now to our original argument that the run is locally safe at the root, we first consider the case where $\mathsf{dom}(a'_r) \notin \mathsf{doms}(S)$ or $\mathsf{dom}(b'_r) \notin \mathsf{doms}(S)$.

In this case $\texttt{sync}(S, o, a, b) = (\mathcal{X}, a, b)$ and it follows immediately from Lemma 13(1) that the run $(S, o, a, b, \mathcal{X}, a, b)$ is safe. To show that it is maximal we show that there is a schema domain conflict at the root. We have that $\mathsf{mdom}(o, a, b) = \mathsf{dom}(a'_r) \notin \mathsf{doms}(S)$ or $\mathsf{mdom}(o, b, a) = \mathsf{dom}(b'_r) \notin \mathsf{doms}(S)$, so $\mathsf{sdomconflict}(S, o, a, b)$ holds. As above, by safety condition (3), the only safe run is $(S, o, a, b, \mathcal{X}, a, b)$, hence it is maximal.

Finally, we consider the case where $\mathsf{dom}(a'_r) \in \mathsf{doms}(S)$ and $\mathsf{dom}(b'_r) \in \mathsf{doms}(S)$. In this case $o' = o'_r$, $a' = a'_r$, and $b' = b'_r$ and since these are assembled as the concatenation of the results of each recursive call, we have $o' \neq \bot$, $a' \neq \bot$, and $b' \neq \bot$ (recall the difference between the empty tree and the missing tree). It follows directly that $a \sim a'$, $b \sim b'$, $a' \sim b'$, $o' \sim a'$, and $o' \sim b'$, which immediately satisfies local safety conditions (1, 2). Local safety condition (3) is satisfied because there cannot be a conflict at the root: there is no local conflict because $o \neq \mathcal{X}$ and $a \sim b$, and there is no schema domain conflict because $\mathsf{mdom}(o, a, b) = \mathsf{dom}(a'_r) \in \mathsf{doms}(S)$ and $\mathsf{mdom}(o, b, a) = \mathsf{dom}(b'_r) \in \mathsf{doms}(S)$. For the final safety condition (4), we show that $a', b' \in S_\bot$. As each sub-run is maximal (and hence, safe), for every $k \in \mathsf{dom}(a')$ we have $a'(k) \in S(k)_\bot$. Also, since $k \in \mathsf{dom}(a')$ we have $a'(k) \neq \bot$ and so $a'(k) \in S(k)$. We also have that $\mathsf{dom}(a') \in \mathsf{doms}(S)$. By Lemma 12, $a' \in S$ and hence $a' \in S_\bot$. By a symmetric argument, we have $b' \in S_\bot$. We conclude that the run is locally safe at the root.

To finish the proof, let $(S, o, a, b, o'', a'', b'')$ be another safe run and let p be a path. We show that both maximality conditions are satisfied by cases on p:

Subcase $p = \bullet$: Both conditions trivially hold as $a' \sim b'$ and $o' \neq \mathcal{X}$.

Subcase $p = k/p'$: The sub-run $(S(k), o(k), a(k), b(k), o''(k), a''(k), b''(k))$ is also safe by definition. We have $a''(p) = a''(k/p') = (a''(k))(p')$, and $b''(p) = b''(k/p') = (b''(k))(p')$.

- To show the first maximality condition, observe that if $a''(p) \sim b''(p)$, then $a''(p) = (a''(k))(p') \sim (b''(k))(p') = b''(p)$. By IH, the run $(o(k), a(k), b(k), o'(k), a'(k), b'(k))$ is maximal. It follows that $(a'(k))(p') \sim (b'(k))(p')$, hence $a'(p) \sim b'(p)$.
- For the second condition, observe that if $o''(p) \neq \mathcal{X}$, then $(o''(k))(p') \neq \mathcal{X}$. By IH, the run $(o(k), a(k), b(k), o'(k), a'(k), b'(k)))$ is maximal and so $(o'(k))(p') \neq \mathcal{X}$. Hence, we have $o'(p) \neq \mathcal{X}$. $\square$