Exploiting Schemas in Data Synchronization

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

Abstract. Increased reliance on optimistic data replication has led to burgeoning interest in tools and frameworks for *synchronizing* disconnected updates to replicated data. We have implemented a generic synchronization framework, called Harmony, that can be 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, state a simple and intuitive specification, and illustrate how it can be used to synchronize trees representing a variety of specific forms of application data, including sets, records, and tuples.

1 Introduction

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

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

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 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 synchronization 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 detail 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.

Our language for lenses has been described in detail elsewhere [7]; in the present paper, our focus is on the synchronization algorithm and the way it uses schema information. The intuition behind 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, this process is actually 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*, in the sense that, if there is *any* well-formed way to propagate a given change from one replica to the other that does not violate schema constraints, then that change *must* be propagated.

Our main technical contribution is a simple one-pass, recursive tree-walking algorithm that does indeed yield results that are maximal in this sense 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. Section 6 illustrates the behavior of the algorithm using a simple address book schema. Related work is discussed in Section 7.

2 Data Model

Internally, Harmony manipulates structured data in an extremely simple form: unordered, edge-labeled trees; richer external formats such as XML are encoded in terms of unordered trees. We chose this simple data model on pragmatic grounds: the reduction in the overall complexity of the Harmony system far outweighs the cost of manipulating ordered data in encoded form.

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 "$\texttt{X} \mapsto \ldots$" denotes a child labeled `X`—e.g., $\{\texttt{Pat} \mapsto \texttt{111-1111},\ \texttt{Chris} \mapsto \texttt{222-2222}\}$. 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 "$\{\texttt{111-1111} \mapsto \{\}\}$."

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 $dom(t)$ for its domain—i.e. the set of the names of its children. The concatenation operator, $\cdot$ is only defined for trees t and t' with disjoint domains; $t \cdot t'$ is the tree mapping n to $t(n)$ for $n \in dom(t)$, to $t'(n)$ for $n \in dom(t')$. When $n \notin dom(t)$, we define $t(n)$ to be $\bot$, the "missing tree." By convention, we take $dom(\bot) = \emptyset$. To represent conflicts during synchronization, we enrich the set of trees with a special pseudo-tree $\mathcal{X}$, pronounced "conflict." We define $dom(\mathcal{X}) = \{n_\mathcal{X}\}$, where $n_\mathcal{X}$ is a special name that does not occur in ordinary trees. We write $\mathcal{T}_\bot$ for $\mathcal{T} \cup \{\bot\}$ and $\mathcal{T}_\mathcal{X}$ for the set of extended trees that may contain $\mathcal{X}$ as a subtree.

A *path* is a sequence of names. We write $\bullet$ for the empty path and p/q for the concatenation of 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 in the obvious way: (1) $t(\bullet) = t$, (2) $t(n/p) = (t(n))(p)$ if $t \neq \mathcal{X}$ and $n \in dom(t)$, (3) $t(n/p) = \bot$ if $t \neq \mathcal{X}$ and $n \notin dom(t)$, and (4) $t(p) = \mathcal{X}$ if $t = \mathcal{X}$.

A tree is included in another tree, written $t \sqsubset t'$, iff any missing or conflicting path in t' is missing in t: $\forall p \in \mathcal{P}.\ (t'(p) = \bot \vee t'(p) = \mathcal{X}) \implies t(p) = \bot$.

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

3 Basics

Harmony's synchronization algorithm takes two[1] replicas $a, b \in \mathcal{T}_\bot$ and a common ancestor $o \in \mathcal{T}_\mathcal{X}$ and yields new replicas in which all non-conflicting updates are merged. Suppose that we have a tree representing a phone book, $o = \{\texttt{Pat} \mapsto \texttt{111-1111},\ \texttt{Chris} \mapsto \texttt{222-2222}\}$. Now suppose we make two replicas of this structure, a and b and separately modify one phone number in each so that $a = \{\texttt{Pat} \mapsto \texttt{111-1111},\ \texttt{Chris} \mapsto \texttt{888-8888}\}$ and $b = \{\texttt{Pat} \mapsto \texttt{999-9999},\ \texttt{Chris} \mapsto \texttt{222-2222}\}$. Synchronization takes these structures and produces a structure $o' = \{\texttt{Pat} \mapsto \texttt{999-9999},\ \texttt{Chris} \mapsto \texttt{888-8888}\}$ that reflects all the changes in a and b with respect to o. We save the final merged state at the end of each synchronization, to use as the o input of the next synchronization.

Loose Coupling Harmony is a *state-based* synchronizer: only the current states of the replicas (plus the remembered state o) are supplied to the synchronizer, rather than the sequence of operations that produced a and b from o. 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 (or *state-based*) approach is that we can use Harmony to synchronize off-the-shelf applications that were implemented without replication in mind. By contrast, many synchronizers manipulate a trace of

[1] We focus on the two-replica case. Our algorithm generalizes straightforwardly to synchronizing n replicas, but the more realistic case of a network of possibly *disconnected* replicas poses additional challenges (see [8] for our progress in this area).

the operations that the application has performed on each replica, and propagate changes by undoing and/or replaying operations. This approach requires 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, we change both Pat's and Chris's phone numbers in a and, in b, 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 o' that incorporates both of the changes to Chris: we have a *delete/modify conflict*. At this point, we must choose between two evils. On one hand, we can weaken users' expectations for the *persistence* of their changes to the replicas—i.e., we can decline to promise that synchronization will never lose or back out any changes that have explicitly been made to either replica. For example, here, we might back out the deletion of Chris: $a' = b' = \{\texttt{Pat} \mapsto \texttt{123-4567}, \texttt{Chris} \mapsto \texttt{888-8888}\}$. 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 letting $a' = \{\texttt{Pat} \mapsto \texttt{123-4567}, \texttt{Chris} \mapsto \texttt{888-8888}\}$ and $b' = \{\texttt{Pat} \mapsto \texttt{123-4567}\}$, and notifying the user of the conflict.[2] In Harmony, we choose persistence.

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 [12], the work of Chawathe et al [4], and FCDP [11]—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 decisions—i.e., it can be difficult for a user to predict how data will be aligned, especially in cases where both replicas have changed significantly. 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 6.

[2] An industrial-strength synchronization tool will not only notify the user of conflicts, but may also assist in bringing the replicas back into agreement by providing graphical views of the differences, applying special heuristics, etc. We omit discussion of this part of the process, focusing on the synchronizer's basic, "unattended" behavior.

An important avenue for future work is hybridizing local and global alignment techniques to combine their advantages.

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 to be 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, is at the root level. The key fields, which uniquely identify a contact, are often drawn from some underlying database:

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

Using the alignment scheme described above, the effect during synchronization will be that entries from the two replicas with the same UID are synchronized with each other. Alternatively, if UIDs are not available, we can synthesize a UID by lifting information out of each record—e.g., we might concatenate the `name` data and use it as the top-level key field: $\{\texttt{Megan:Smith} \mapsto \{\texttt{home} \mapsto \texttt{555-6666}\},\ \texttt{Pat:Jones} \mapsto \{\texttt{home} \mapsto \texttt{555-2222}\}\}$.

It is unlikely, however, that the address book will be represented concretely (e.g., as an XML document) using either of these formats. To bridge this gap, the Harmony system includes a domain-specific language for writing bi-directional transformations [7], which we call *lenses*. 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* [2]). Lenses also facilitate synchronization of heterogeneous formats. Since each replica is passed through a lens both before and after synchronization, it does not much matter if the replicas are represented in the same format or not. We can apply a different lens on each side to bring replicas stored using different concrete representations into the same format for synchronization.

4 The Role of Schemas

We impose two core requirements on synchronization, which we call *safety* and *maximality* and describe informally here (the long version has precise definitions).

Safety The safety requirement encompasses four sanity checks: (1) a synchronizer must not "back out" any changes made at a replica since the last synchronization (because we favor persistence over convergence); (2) it should only copy data from one replica to the other, never "make up" content on its own; (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.

Schema Conflicts Our algorithm (unlike other state-based synchronizers) is designed to preserve structural invariants. As an example of how schema invariants can be broken, consider a run of the algorithm sketched above where $o = \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{333-4444} \mapsto \{\}\}\}\}$, $a = \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{111-2222} \mapsto \{\}\}\}\}$,

and $b = \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{987-6543} \mapsto \{\}\}\}\}$. 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 $a' = b' = \{\texttt{Pat} \mapsto \{\texttt{Phone} \mapsto \{\texttt{111-2222} \mapsto \{\},\ \texttt{987-6543} \mapsto \{\}\}\}\}$. But this would be wrong. Pat's phone number was *changed* in different ways in the two replicas: what's wanted is a conflict. If the phonebook schema only allows a single number per person, then the new replica is not well formed!

We avoid these situations by equipping the synchronizer with knowledge of the intended schema of the structures it is synchronizing. It can then simply signal a conflict (leaving its inputs unchanged) whenever it sees that merging the changes at a particular point will lead to an ill-formed structure.

Locality and Schemas Because alignment in our algorithm is local, we cannot expect the algorithm to enforce global invariants expressed by arbitrary schemas; we need a corresponding restriction to schemas that permits them to express only local constraints on structure. As an example of a schema that expresses a *non*-local invariant, consider the following set of trees: $\{\{\}, \{\texttt{n} \mapsto \texttt{x}, \texttt{m} \mapsto \texttt{x}\}, \{\texttt{n} \mapsto \texttt{y}, \texttt{m} \mapsto \texttt{y},\}, \{\texttt{n} \mapsto \{\texttt{x}, \texttt{y}\}, \texttt{m} \mapsto \texttt{y}\}, \{\texttt{n} \mapsto \texttt{x}, \texttt{m} \mapsto \{\texttt{x}, \texttt{y}\}\}\}$. Now consider synchronizing two trees from this set with respect to the empty archive $o = \{\}$, with $a = \{\texttt{n} \mapsto \texttt{x},\ \texttt{m} \mapsto \texttt{x}\}$, and $b = \{\texttt{n} \mapsto \texttt{y},\ \texttt{m} \mapsto \texttt{y}\}$. A local synchronization algorithm that aligns by name wants to recursively synchronize the subtrees under `n` and `m`. However, it is not clear what *schema* we should 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 actually $a' = \{\texttt{n} \mapsto \texttt{x},\ \texttt{m} \mapsto \{\texttt{x},\ \texttt{y}\}\}$ 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 with a non-backtracking algorithm.

The global invariant expressed by this schema—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 be validly "transplanted" to the same location in any other tree in the schema. This restriction ensures that the sub-schema used to synchronize a single child is consistent across the schema; i.e., the set of trees that may validly appear under a child only depends on the path from the root to the node and does not depend on the presence (or absence) of other parts of the tree.

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

Next, we define what it means to transplant a subtree from one tree to another at a given path. Let t be a tree and p a path such that $t(p) \in \mathcal{T}$. We define the *update* of t at p with t', written $t[p \mapsto t']$, inductively on the structure of p as: $t[\bullet \mapsto t'] = t', t[n/p \mapsto t'] = \{n \mapsto t(n)[p \mapsto t'], m \mapsto t(m) \mid m \in dom(t) \setminus \{n\}\}$. Now, a schema S is *path consistent* if, whenever t and t' are in S, it is the case that, 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 $p \in \mathcal{P}$, we have $t(p) \neq \bot \ \wedge \ t'(p) \neq \bot \implies t[p \mapsto t'(p)] \in S$.

Maximality Of course, safety by itself is too weak: the trivial algorithm that always returns both replicas unchanged is perfectly safe! We therefore say that a run of a synchronizer is *maximal* just in case it propagates as many changes as it safely can. The specification for our synchronization algorithm is that every run must be both safe and maximal.

This brings us to one final complication that arises in schema-aware synchronization algorithms: for some inputs, there *aren't any* safe, maximal runs belonging to the schema. Consider a run of a synchronizer on the inputs $o = \{\mathtt{v}\}$, $a = \{\mathtt{w}, \mathtt{y}, \mathtt{z}\}$, and $b = \{\mathtt{w}, \mathtt{x}\}$, with respect to the schema $\{\{\mathtt{v}\}, \{\mathtt{w}, \mathtt{x}\}, \{\mathtt{w}, \mathtt{x}, \mathtt{y}\}, \{\mathtt{w}, \mathtt{x}, \mathtt{z}\}, \{\mathtt{w}, \mathtt{y}, \mathtt{z}\}.\}$ On the b side, there are three safe results belonging to the schema, $\{\mathtt{w}, \mathtt{x}\}$, $\{\mathtt{w}, \mathtt{x}, \mathtt{y}\}$, and $\{\mathtt{w}, \mathtt{x}, \mathtt{z}\}$, but none is maximal. Notice that, since $\{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$ does not belong to the schema, we cannot include both `y` and `z` in b' (without backing out the addition of `x`). Indeed, for every safe choice of b', there is a path p where $b'(p) \neq a'(p)$, but, for a different choice of b', the trees at those paths are equal.

To ensure that synchronization always has a maximal result, we stipulate that a *schema domain conflict* occurs whenever propagating *all* of the (otherwise non-conflicting) additions and deletions of children at a node yields an ill-formed result. On the above trees, our algorithm yields a schema domain conflict at the root since it cannot add `y` and `z` to a'.

5 Algorithm

The synchronization algorithm is depicted in Figure 1. Its structure is as follows: we first check for trivial cases (replicas being equal to each other or unmodified), then we check for *delete/create conflicts*, and in the general case we recurse on each child label and check 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 needs to construct a new archive. Its calculation is straightforward: we use the synchronized version at every path where the replicas agree and insert a conflict marker $\mathcal{X}$ at paths where they conflict.

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

$sync(S, o, a, b) =$
  if $a = b$ then(a, a, b) – *equal replicas: done*
  else if $a = o$ then (b, b, b) – *no change to a*
  else if $b = o$ then (a, a, a) – *no change to b*
  else if $o = \mathcal{X}$ then (o, a, b) – *unresolved conflict*
  else if $a = \bot$ and $b \sqsubset o$ then (a, a, a) – *a deleted more than b*
  else if $a = \bot$ and $b \not\sqsubset o$ then $(\mathcal{X}, a, b)$ – *delete/create conflict*
  else if $b = \bot$ and $a \sqsubset o$ then (b, b, b) – *b deleted more than a*
  else if $b = \bot$ and $a \not\sqsubset o$ then $(\mathcal{X}, a, b)$ – *delete/create conflict*
  else – *proceed recursively*
    let $(o'(k), a'(k), b'(k)) = sync(S(k), o(k), a(k), b(k))$
        $\forall k \in dom(a) \cup dom(b)$
    in if $(dom(a') \notin doms(S))$ or $(dom(b') \notin doms(S))$
      then $(\mathcal{X}, a, b)$ – *schema domain conflict*
      else (o', a', b')

Fig. 1. Synchronization Algorithm

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 one replica is missing, then we check whether all the changes in the other replica are also deletions; if so, we consider the larger deletion (throwing away the whole tree at this point) as subsuming the smaller; otherwise, we have a *delete/create conflict* and we simply return the original replicas.

Finally, in the general case, the algorithm recurses: for each k in the domain of either current replica, we call *sync* with the corresponding subtrees, $o(k)$, $a(k)$, 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.

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

6 Case Study: Address Books

We now present a brief case study, illustrating how schemas can be used to guide the behavior of our generic synchronizer on trees of realistic complexity. The examples use an address book schema loosely based on the vCard standard.

Schemas We begin with a concrete notation for writing schemas. Schemas are given by mutually recursive equations of the form `X = S`, where `S` is generated by the following grammar: `S ::= {}` | `n[S]` | `!(F)[S]` | `*(F)[S]` | `S,S` | `S|S`. Here `n` ranges over names in $\mathcal{N}$ and `F` ranges over finite sets of names. 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* not in `F`, where the subtree under that child is in `S`; the other wildcard schema, `*(F)[S]` denotes the set of trees with *any number of children* not in `F` where the subtree under each child is in `S`. The set of trees described by $\texttt{S}_1\texttt{|}\texttt{S}_2$ is the union of the sets described by $\texttt{S}_1$ and $\texttt{S}_2$, while $\texttt{S}_1\texttt{,}\texttt{S}_2$ denotes the set of trees $t_1 \cdot t_2$ where t_1 belongs to $\texttt{S}_1$ and t_2 to $\texttt{S}_2$. 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]`.

All the schemas we write are path consistent. This can be checked syntactically: if a name appears twice as a child of a node, like `m` in `m[X],n[Y]|m[X],o[Z]`, the schemas of the associated subtrees are textually identical.

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):

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

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

The schema `C` describes both sorts of contacts (using some sub-schemas that we will define below): `C = name[N],work[V],home?[V],org[O],email[E] | name[N],work?[V],home[V],org?[O],email[E].` The trees appearing under the `home` and `work` children represent simple string values—i.e., trees with a single child leading to the empty tree; they belong to the `V` schema, `V = ![{}].` The `name` edge leads to a tree with a record-like structure containing mandatory `first` and `last` fields and an optional `other` field. The `first` and `last` fields lead to values belonging to the `V` schema. The `other` field leads to a list of alternate names such as middle names or nicknames, stored (for the sake of the example) in some particular order. Because our actual trees are unordered, we use a standard "cons cell" representation to encode ordered lists. The list $[t_1; \ldots; t_n]$ is encoded as the tree $\{\texttt{head} \mapsto \texttt{t}_1\ ,\ \texttt{tail} \mapsto \{\ldots \mapsto \{\texttt{head} \mapsto \texttt{t}_n\ ,\ \texttt{tail} \mapsto \texttt{nil}\}\ldots\}\}$. Using this representation of lists, the `N` schema is defined straightforwardly as: `N = first[V],other?[VL],last[V],` where `VL` is a schema that describes lists of values (encoded as trees): `VL = head[V],tail[VL] | nil[{}].` The email address data for a contact is either a single value, or a set of addresses with one distinguished "preferred" address. The `E` schema describes these structures using a union of a wildcard to represent single values (which excludes `pref` and `alts` to

ensure path consistency) and a record-like structure with fields `pref` and `alts` to represent sets of addresses: `E = !(pref, alts)[{}] | pref[V],alts[VS],` where `VS = *[{}]` describes the trees that may appear under `alts`—bushes with any number of children where each child leads to the empty tree. These bushes are a natural encoding of sets of values as trees. Finally, organization information is represented by a structure with `orgname` and `orgunit` fields, each leading to a value, as described by this schema: `O = orgname[V],orgunit[V].`

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],` each showing a different way in which schema-ignorant synchronization can produce mangled results. In each case, the archive, o, is the tree shown above.

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

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

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

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

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

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

As a final example, consider changes to the list of names stored at the path `name/other`. Suppose that a removes both `Liz` and `Jo`, but b only removes `Jo`: $o(\texttt{name/other}) = [\texttt{Liz};\ \texttt{Jo}]$, $a(\texttt{name/other}) = []$, 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 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 any 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 $\mathit{doms}(\texttt{C})$. 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 $\mathit{doms}(\texttt{V})$. In the example involving the tree under `email`, the algorithm tests if the domain $\{\texttt{meg@s.com}, \texttt{pref}, \texttt{alts}\}$ is in $\mathit{doms}(\texttt{E})$. Finally, in the example where both replicas updated the list of `other` names, it tests whether $\{\texttt{tail}, \texttt{nil}\}$ is in $\mathit{doms}(\texttt{VL})$. 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.

In the remainder of the section, we further explore the strengths (and weaknesses) of our approach by examining the behavior of our algorithm on the structures that are 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 to 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. That is, the algorithm enforces *atomic* updates to values.

Sets Sets can be represented as bushes—nodes with many children, each labeled with the key of an element in the set— e.g., for sets of values, this structure is described by 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: $o = \{\texttt{meg@s.com}\}$, $a = \{\texttt{ms@c.edu}, \texttt{meg.smith@cs.c.edu}\}$, and $b = \{\texttt{meg@s.com}, \texttt{meg.smith@cs.c.edu}\}$ 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 $a' = b' = \{\texttt{ms@c.edu}, \texttt{meg.smith@cs.c.edu}\}$, as expected.

Records Two sorts of record structures appear in the address book schema. The simplest records, like the one for organization data (`orgname[V],orgunit[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 preceeding 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.

Lists Lists present special challenges, because we would like the algorithm to detect updates both to elements and to their relative position. On lists, our local alignment strategy matches up list elements by *absolute* position, leading to surprising results on some inputs. We illustrate the problem and propose a more sophisticated encoding of lists that reduces the chances of confusion.

On many runs of the synchronizer, updates to lists can be successfully propagated from one replica to the other. If either replica is identical to the archive, then the algorithm trivially copies all of the changes from the other replica. Or, if both replicas are not equal to the archive but each replica modifies a disjoint subset of the elements of the list (and leaves the spine of the list intact), then the synchronizer can merge the changes successfully. There are some inputs, however, where synchronizing lists using the local alignment strategy and simple cons cell encoding produces strange results. Consider a run of the synchronizer on the following trees: $o = [\texttt{Liz}; \texttt{Jo}]$, $a = [\texttt{Jo}]$ 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 somewhat surprising: $[\texttt{Jo}; \texttt{Joanna}]$. The algorithm does not recognize that `Jo` and `Joanna` should be aligned. Instead, it aligns pieces of the list by absolute position, matching `Jo` with `Liz` and `nil` with $[\texttt{Joanna}]$.

It is not surprising that our algorithm doesn't have an intuitive behavior when its inputs are lists. In general, detecting changes in relative position in a list requires global reasoning but our algorithm is essentially local. In order to avoid these problematic cases, we can use an alternative schema, which we call the *keyed list schema*, for lists whose relative order matters. Rather than embedding the elements under a spine of cons cells, one can lift up the value at each position into the spine of the list. For example, in the extended encoding, the list a from

above is represented as the tree $a = \{\texttt{Jo} \mapsto \{\texttt{head} \mapsto \{\}, \texttt{tail} \mapsto \texttt{nil}\}\}$.[3] The schema for keyed lists of values is: `KVL = !(nil)[head[{}],tail[KVL]] | nil.` During synchronization, elements of the list are identified by the value above each cons cell; synchronization proceeds until a trivial case applies (unchanged or identical replicas), or when the two replicas disagree on the domain of an element, resulting in a schema domain conflict. In the problematic example, the algorithm terminates with a conflict at the root. Keyed lists combine an alternate representation of lists with an appropriate schema to ensure that the local algorithm has reasonable (if conservative) behavior.

Conclusion The examples in this section demonstrate that schemas are a valuable 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, we can tune the behavior of the generic algorithm to work well with a variety of structures. While building demos using our prototype implementation, we have found that this works well with rigidly structured data (e.g., values and records) and unstructured data (e.g., sets of values), but so far has limited utility when used with ordered and semi-structured data (e.g., lists and documents). In the future, we hope to extend our algorithm to better handle ordered data.

7 Related Work

In the taxonomy of optimistic replication strategies in the survey by Saito and Shapiro [22], Harmony is a multi-master state-transfer system, recognizing sub-objects and manually resolving conflicts. Harmony is further distinguished by some distinctions not covered in that survey: it is generic, loosely coupled from applications, able to synchronize heterogeneous representations, and is usable both interactively and *unsupervised*. Supporting unsupervised runs (where Harmony does as much work as it can, and leaves conflicts for later) requires our synchronizer's behavior to be intuitive and easy to predict.

IceCube [10] is a generic operation-based reconciler that is parameterized over a specific algebra of operations appropriate to the application data being synchronized and by a set of syntactic/static and semantic/dynamic ordering constraints on these operations. Molli et al [15], 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. Bengal [6] records operations to avoid scanning the entire replica during update detection. Like Harmony, Bengal is a loosely-coupled synchronizer. It can extend any commercial database system that uses OLE/COM hooks to support optimistic replication. However, it is not generic because it only supports databases, it is not heterogeneous because reconciliation can only occur between replicas of the same database, and it requires users to write *conflict resolvers* if they want to avoid manually resolving conflicts. FCDP [11] is a generic, state-based reconciler parameterized by

[3] For keyed lists of values, we could drop the child `head`, which always maps to the empty tree. However, we can also form keyed lists of arbitrary trees, not just values.

ad-hoc translations from heterogeneous concrete representations to XML and back again. There is no formal specification and reconciliation takes place at "synchronization servers" that are assumed to be more powerful machines permanently connected to the network. FCDP fixes a specific semantics for ordered lists—particularly suited for document editing. This interpretation may sometimes be problematic, as described in the long version of this paper. File system synchronizers (such as [23, 16, 1, 18]) and PDA synchronizers (such as Palm's HotSync), are not generic, but they do generally share Harmony's state-based approach. An interesting exception is DARCS [21], a hybrid state-/operation-based revision control system built on a "theory of patches."

Harmony, unlike many reconcilers, does not guarantee convergence in the case of conflicts. Systems such as Ficus [19], Rumor [9], Clique [20], Bengal [6], and TAL/S5 [15] converge by making additional copies of primitive objects that conflict and renaming one of the copies. CVS embeds markers in the bodies of files where conflicts occurred. In contrast, systems such as Harmony and IceCube [10] will not reconcile objects affected by conflicting updates.

Harmony's emphasis on schema-based pre-alignment is influenced by examples we have found in the context of data integration where heterogeneity is a primary concern. Alignment, in the form of schema-mapping, has been frequently used to good effect (c.f. [17, 14, 3, 5, 13]). 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 [1] (which has almost trivial alignment) and CVS, is the comparative alignment strategy (based on the standard Unix tool `diff3`) used by CVS. At this stage, Harmony's core synchronization algorithm is deliberately simplistic, particularly with respect to ordered data. As we develop an understanding of how to integrate more sophisticated alignment algorithms in a generic and principled way, we hope to incorporate them into Harmony. Of particular interest are `diff3` and its XML based descendants: Lindholm's 3DM [12], the work of Chawathe et al [4], and FCDP [11].

References

1. S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, Oct. 1998. Full version available as Indiana University CSCI technical report #507, April 1998.
2. F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 6(4):557–575, 1981.
3. C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *ICDT'99*, 1999.
4. S. S. Chawathe, A. Rajamaran, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on the management of Data*, pages 493–504, Montreal, Quebec, 1996.
5. A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, 2001.

6. T. Ekenstam, C. Matheny, P. L. Reiher, and G. J. Popek. The Bengal database replication system. *Distributed and Parallel Databases*, 9(3):187–210, 2001.
7. J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, 2005.
8. M. B. Greenwald, S. Khanna, K. Kunal, B. C. Pierce, and A. Schmitt. Agreement is quicker than domination: Conflict resolution for optimistically replicated data. Submitted for publication; available electronically, 2005.
9. R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *Proceedings of the ER '98 Workshop on Mobile Data Access*, pages 254–265, 1998.
10. A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of diverging replicas. In *proceedings of the 20th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '01)*, Aug. 26-29 2001. Newport, Rhode Island.
11. M. Lanham, A. Kang, J. Hammer, A. Helal, and J. Wilson. Format-independent change detection and propoagation in support of mobile computing. In *Proceedings of the XVII Symposium on Databases (SBBD 2002)*, pages 27–41, October 14-17 2002. Gramado, Brazil.
12. T. Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *Proceedings of MobiDE '03*, pages 93–97, September 19 2003. San Diego, CA.
13. J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *The VLDB Journal*, pages 49–58, 2001.
14. T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB'98*, 1998.
15. P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proceedings of ACM Group 2003 Conference*, November 9–12 2003. Sanibel Island, Florida.
16. T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 11(1), December 1997.
17. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
18. N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. In *Proceedings of the 8th European Software Engineering Conference*, pages 175–185. ACM Press, 2001.
19. P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the ficus file system. In *USENIX Summer Conference Proceedings*, pages 183–195, 1994.
20. B. Richard, D. M. Nioclais, and D. Chalon. Clique: a transparent, peer-to-peer collaborative file sharing system. In *Proceedings of the 4th international conference on mobile data management (MDM '03)*, Jan. 21-24 2003. Melbourne, Australia.
21. D. Roundy. DARCS revision control system, 2004. `http://www.abridgegame.org/darcs/`.
22. Y. Saito and M. Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, HP Laboratories Palo Alto, Feb. 8 2002.
23. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, C-39(4):447–459, Apr. 1990.