

Harmony: A Synchronization Framework For Tree-Structured Data

Michael B. Greenwald, Owen Gunden,
Jonathan T. Moore, Benjamin C. Pierce,
Alan Schmitt

University of Pennsylvania

September, 2003

Optimistic Replication

- Many copies of data, spread across many machines
- Any copy may be updated at any time
- Hosts occasionally **synchronize** (or “reconcile”) their states



Advantages of Optimism

- **Availability:** users can work while disconnected
- **Scalability:** no “hot spots” for writes
- **Quality control:** updates can be “curated” before being allowed into a replica
- **Visibility/atomicity control:** a set of updates can be “kept local” until ready, then propagated to other machines (cf. CVS)



Challenges of Optimism

Pragmatic issues:

- How to make sure updates get propagated in a timely manner (while dealing with failures, avoiding swamping the network or using too much local storage, etc.)?
⇒ Interesting when there are many replicas

Semantic issues:

- Precisely what does it mean to “synchronize” replicas?
⇒ Interesting when replicated data has nontrivial internal structure



Synchronizing States vs. Operations

Two basic approaches to semantics of synchronization:

- **Operation-based** synchronizers are given access to complete traces of all the **operations** performed on each replica of the data.
Examples: Bayou, IceCube
- **State-based** synchronizers are given just the **states** of the replicas at particular moments in time.
Examples: Unison, Harmony

Tradeoffs

Operation-based:

- **Pro:** temporal sequencing of operations on each replica visible to synchronizer
- **Pro:** operations can be chosen to encode high-level application semantics
- **Con:** require relatively tight coupling with applications
⇒ Appropriate for “synchronization middleware”

State-based:

- **Con:** less information available at sync time
- **Pro:** applications need not be “synchronization aware”
⇒ Appropriate for loosely coupled architectures

Caveat

The state-based / operation-based distinction is a bit slippery and various hybrids are possible. E.g...

- Can build a state-based system with an operation-based core by diffing previous and current states to obtain a hypothetical (typically, minimal) sequence of operations

Caveat

The state-based / operation-based distinction is a bit slippery and various hybrids are possible. E.g...

- Can build a state-based system with an operation-based core by diffing previous and current states to obtain a hypothetical (typically, minimal) sequence of operations
But... this involves complex heuristics, which can be difficult for users to understand

The Harmony Project

Research goal:

Build a (pure) state-based synchronizer for structured data stored as XML documents

The Harmony Project

Research goal:

Build a (pure) state-based synchronizer for structured data stored as XML documents

Simplification:

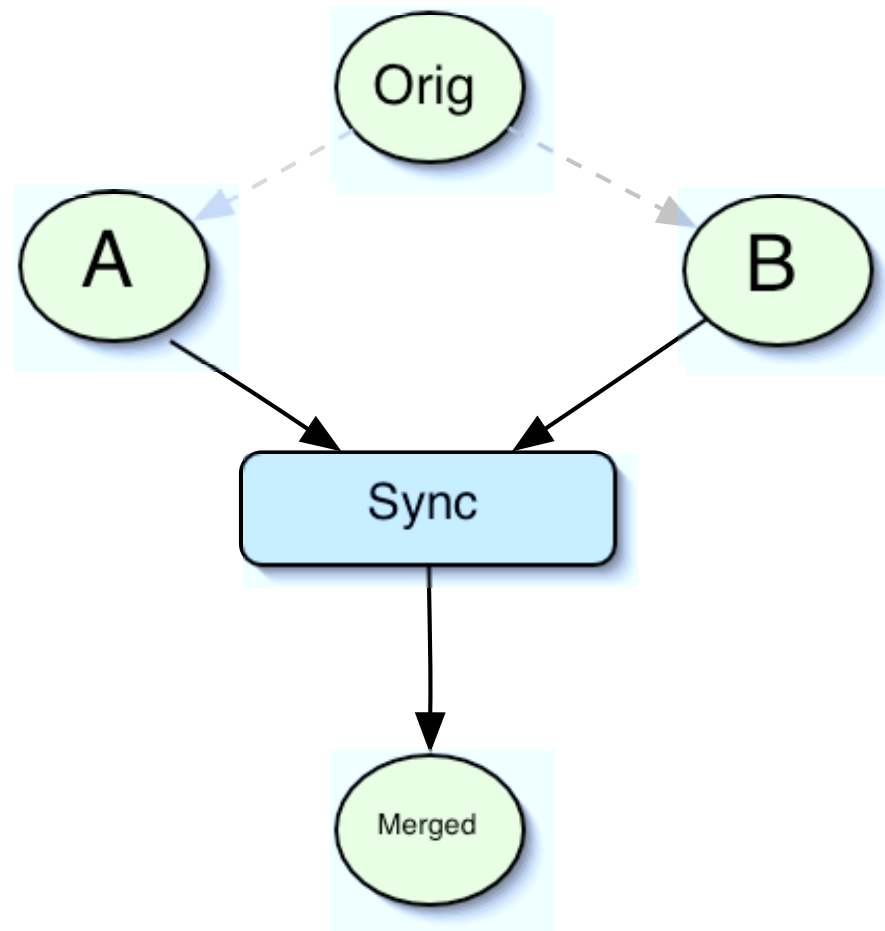
For this talk, I'll focus on the 2-replica case

Outline of talk

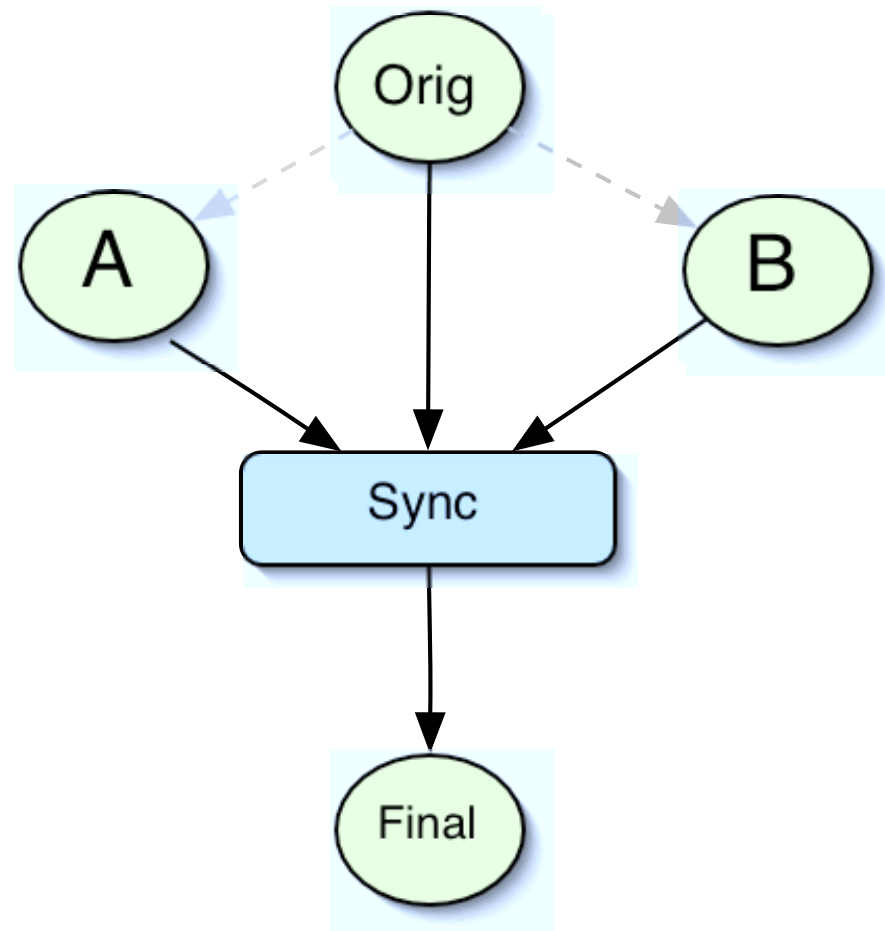
- Architecture
- Data model
- Synchronization algorithm
- Tree transformation language
- Applications

Architecture

Architecture (first cut)



Better Architecture



Conflicts

There will be times when the synchronizer cannot merge the changes to the replicas.

on replica A we create a file x with contents 5

on replica B we create a file x with contents 7

What to do?

Conflicts

- A state-based synchronizer tries to make the replicas “more similar”

So it is natural to deal with conflicts by allowing replicas to remain different after synchronization

(Note that this never backs out user changes, but does not always achieve convergence.)



Conflicts

- A state-based synchronizer tries to make the replicas “more similar”

So it is natural to deal with conflicts by allowing replicas to remain different after synchronization

(Note that this never backs out user changes, but does not always achieve convergence.)

- By contrast, an operation-based synchronizer tries to construct a common sequence of operations

Thus, it is natural to deal with conflicts by omitting conflicting operations

This always achieves a common final state, but sometimes backs out user changes



Conflicts

- A state-based synchronizer tries to make the replicas “more similar”

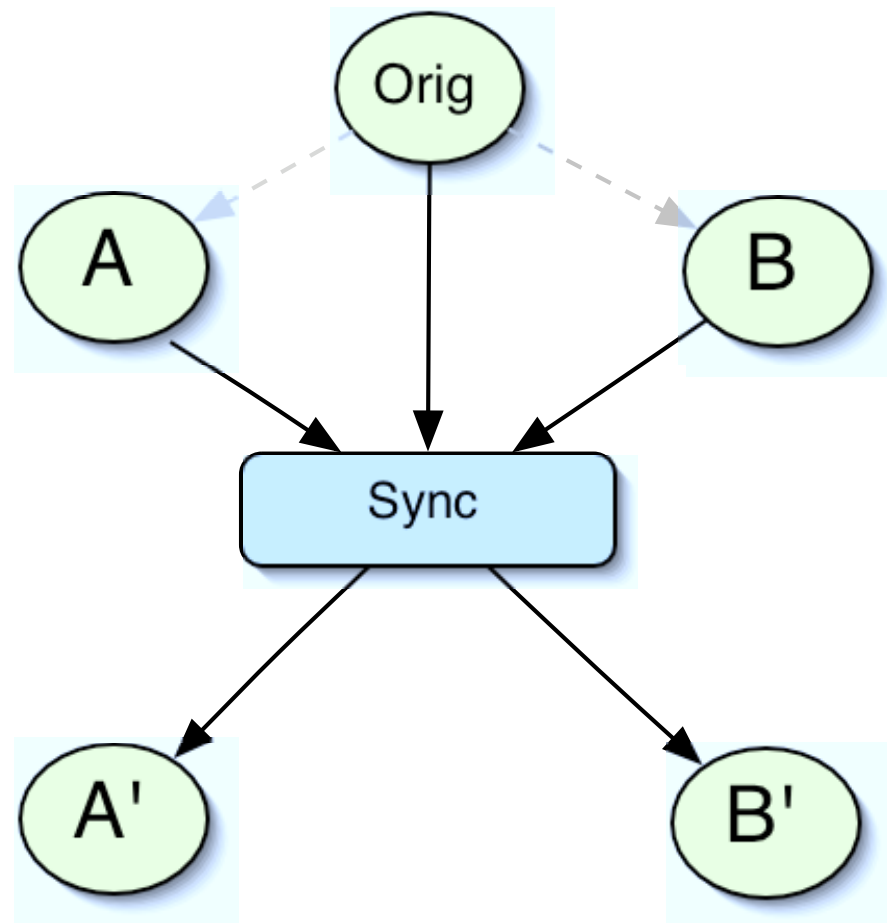
So it is natural to deal with conflicts by allowing replicas to remain different after synchronization

(Note that this never backs out user changes, but does not always achieve convergence.) **N.b.: not forced...**

- By contrast, an operation-based synchronizer tries to construct a common sequence of operations

Thus, it is natural to deal with conflicts by omitting conflicting operations

This always achieves a common final state, but sometimes backs out user changes



Abstraction

Problem: The “concrete representation” of data as XML is not always appropriate for synchronization.

```
Orig = <phonedb>
    <entry> <Name>Pat</> <Phone>333-4444</> </entry>
    <entry> <Name>Chris</> <Phone>888-9999</> </entry>
</phonedb/>
```

```
A    = <phonedb>
    <entry> <Name>Chris</> <Phone>555-6666</> </entry>
    <entry> <Name>Pat</> <Phone>333-4444</> </entry>
</phonedb/>
```

```
B    = <phonedb>
    <entry> <Name>Pat</> <Phone>111-2222</> </entry>
    <entry> <Name>Chris</> <Phone>888-9999</> </entry>
</phonedb/>
```



Abstraction

Problem: The “concrete representation” of data as XML is not always appropriate for synchronization.

```
Orig = <phonedb>
    <entry> <Name>Pat</> <Phone>333-4444</> </entry>
    <entry> <Name>Chris</> <Phone>888-9999</> </entry>
</phonedb/>
```

```
A    = <phonedb>
    <entry> <Name>Chris</> <Phone>555-6666</> </entry>
    <entry> <Name>Pat</> <Phone>333-4444</> </entry>
</phonedb/>
```

```
B    = <phonedb>
    <entry> <Name>Pat</> <Phone>111-2222</> </entry>
    <entry> <Name>Chris</> <Phone>888-9999</> </entry>
</phonedb/>
```



Abstraction

Solution: Transform concrete representations into a more suitable abstract form (in an application-specific way) before synchronizing

$$\text{Orig} = \begin{cases} \text{Pat} \mapsto 333-4444 \\ \text{Chris} \mapsto 888-9999 \end{cases}$$

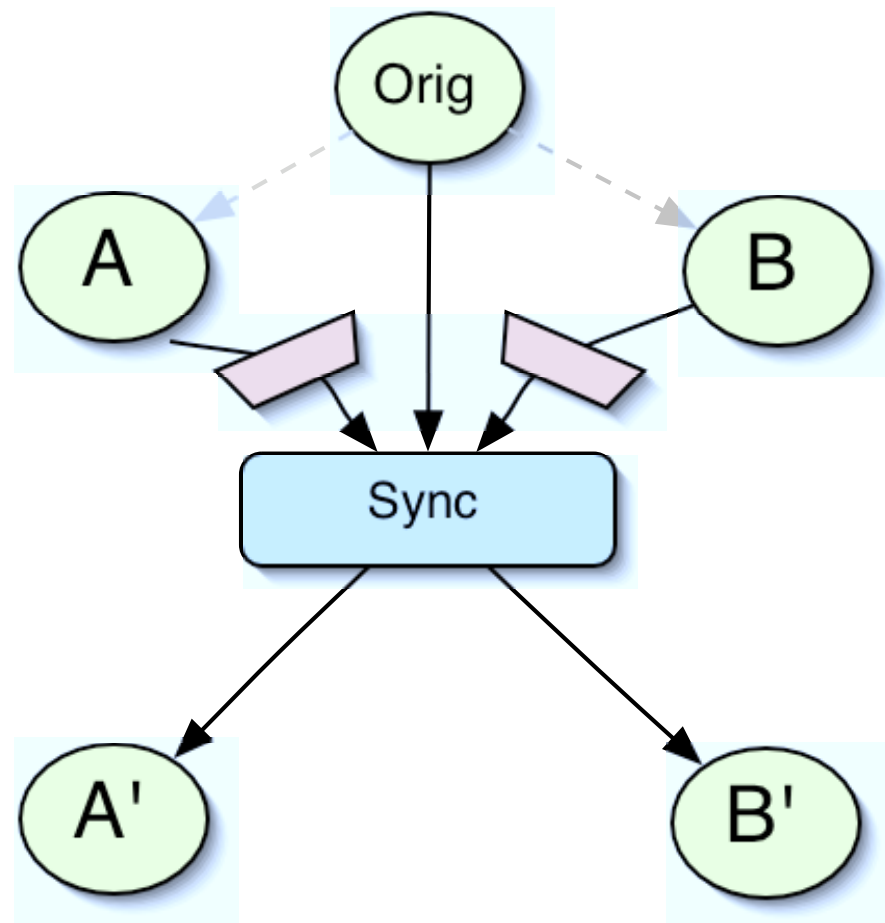
$$\text{A} = \begin{cases} \text{Chris} \mapsto 888-9999 \\ \text{Pat} \mapsto 555-6666 \end{cases}$$

$$\text{B} = \begin{cases} \text{Pat} \mapsto 333-4444 \\ \text{Chris} \mapsto 111-2222 \end{cases}$$

(Trees are drawn sideways; children are unordered.)



Architecture with “Lenses”

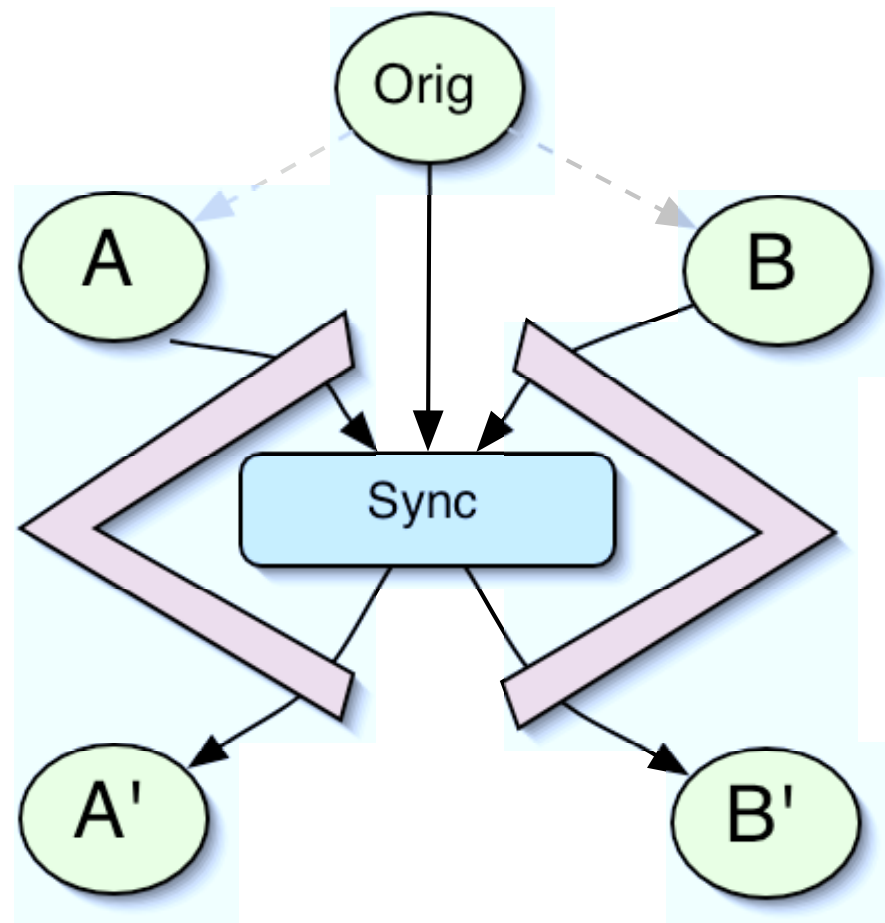


Concretion

Of course, after synchronization, we want our data back in its original concrete form.

I.e., the abstraction function must be “wrapped around” both sides of the core synchronization engine, mapping structures from concrete to abstract and back again.

Final Architecture



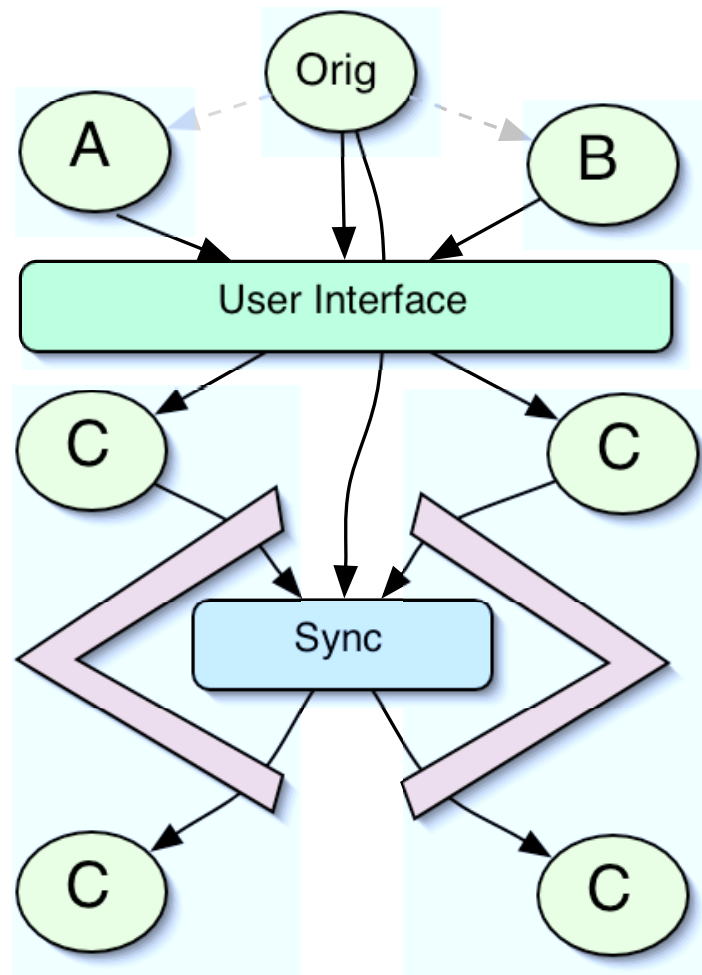
User Interface

When discussing the core synchronization algorithm, it is convenient to ignore user interface issues: in case of a conflict, the output replicas are simply left unchanged from the inputs.

The user interface is regarded as a separate tool that takes the same inputs as the synchronizer, reports conflicts, and allows the user to bring the replicas into a consistent state.

The synchronizer then simply notes that the replicas are equal.

Final Architecture (with UI)



Data Model

Trees

Harmony's core data model is the simplest we could think of — unordered, edge-labeled trees with all children of a node labeled differently.

(I.e., each tree is a partial function from labels to subtrees.)

$$\left\{ \begin{array}{l} \text{Pat} \mapsto \left\{ \begin{array}{l} \text{Phone} \mapsto \{ 333-4444 \mapsto \{ \\ \text{URL} \mapsto \{ \text{http://pat.com} \mapsto \{ \end{array} \right. \\ \\ \text{Chris} \mapsto \left\{ \begin{array}{l} \text{Phone} \mapsto \{ 888-9999 \mapsto \{ \\ \text{URL} \mapsto \{ \text{http://chris.org} \mapsto \{ \end{array} \right. \end{array} \right.$$

Lists

The list

$$[v_1 \dots v_n]$$

is represented by the tree

$$\left\{ \begin{array}{l} *h \mapsto v_1 \\ *t \mapsto \left\{ \begin{array}{l} *h \mapsto v_2 \\ *t \mapsto \left\{ \dots \mapsto \left\{ \begin{array}{l} *h \mapsto v_n \\ *t \mapsto \{ \end{array} \right. \end{array} \right. \end{array} \right. \end{array} \right.$$

XML

The XML element

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

is represented by the tree

$$\left\{ \begin{array}{l} \text{tag} \mapsto \left\{ \begin{array}{l} \text{attr1} \mapsto \left\{ \text{val1} \mapsto \{ \right. \\ \vdots \\ \text{attrm} \mapsto \left\{ \text{valm} \mapsto \{ \right. \\ \text{*content} \mapsto \left[\begin{array}{l} \langle \text{subelt1} \rangle \\ \vdots \\ \langle \text{subeltn} \rangle \end{array} \right] \end{array} \right. \end{array} \right.$$

Synchronization

Definitions

- **path** = a sequence of names
- **contents** of a path = either a tree or MISSING
- **change** (in a replica) = a path whose contents are different from the original (last synchronized) tree
- **conflict** = path that has been deleted (changed from some tree to MISSING) in one replica and one of whose descendants has changed in the other

Specification

A good synchronizer should...

1. Never overwrite changes
2. Never “make up” contents
3. Stop at conflicting paths (leaving replicas in their current states)
4. Propagate as many changes as possible without violating above rules

Synchronization Algorithm (first try)

```
sync(O, A, B) =  
  if A = B then (A,B)           -- equal replicas: done  
  else if A = 0 then (B,B)      -- no change to A: propagate B  
  else if B = 0 then (A,A)      -- no change to B: propagate A  
  else if A = MISSING then (A,B) -- delete/modify conflict  
  else if B = MISSING then (A,B) -- delete/modify conflict  
  else                           -- proceed recursively  
    for each child k, let  
      (Ak,Bk) = sync(O(k), A(k), B(k)) in  
    let A' = { k -> Ak } in  
    let B' = { k -> Bk } in  
    (A',B')
```

Refinement: Atomicity

Problem: There are situations where this synchronization algorithm can create ill-formed or nonsensical trees.

for example...

Refinement: Atomicity

Synchronizing two trees representing filesystems...

$$A = \left\{ \text{dir} \mapsto \begin{cases} \text{f} \mapsto \left\{ \text{file} \mapsto \left\{ \begin{matrix} 12345 \mapsto \{ \\ \end{matrix} \right. \\ \text{g} \mapsto \left\{ \text{file} \mapsto \left\{ 22 \mapsto \{ \right. \right. \end{cases} \right.$$

$$B = \left\{ \text{dir} \mapsto \begin{cases} \text{f} \mapsto \left\{ \text{file} \mapsto \left\{ \begin{matrix} 67890 \mapsto \{ \\ \end{matrix} \right. \\ \text{g} \mapsto \left\{ \text{file} \mapsto \left\{ 22 \mapsto \{ \right. \right. \end{cases} \right.$$

...yields a non-filesystem:

$$\left\{ \text{dir} \mapsto \begin{cases} \text{f} \mapsto \left\{ \text{file} \mapsto \left\{ \begin{matrix} 12345 \mapsto \{ \\ 67890 \mapsto \{ \\ \end{matrix} \right. \\ \text{g} \mapsto \left\{ \text{file} \mapsto \left\{ 22 \mapsto \{ \right. \right. \end{cases} \right.$$



Refinement: Atomicity

Solution:

Mark nodes **ATOMIC** if their children should not be “mixed” during synchronization.

$$\left\{ \begin{array}{l} \text{dir} \mapsto \left\{ \begin{array}{l} \text{f} \mapsto \left\{ \begin{array}{l} \text{file} \mapsto \left\{ \begin{array}{l} 12345 \mapsto \{ \\ \text{ATOMIC} \mapsto \{ \end{array} \right. \\ \text{ATOMIC} \mapsto \{ \end{array} \right. \\ \text{g} \mapsto \left\{ \begin{array}{l} \text{file} \mapsto \left\{ \begin{array}{l} 22 \mapsto \{ \\ \text{ATOMIC} \mapsto \{ \end{array} \right. \\ \text{ATOMIC} \mapsto \{ \end{array} \right. \end{array} \right. \\ \text{ATOMIC} \mapsto \{ \end{array} \right. \end{array} \right.$$



Final Specification

A good synchronizer should...

1. Never overwrite changes
2. Never “make up” contents
3. Stop at conflicting paths (leaving replicas in their current states)
4. Leave the domain of each node marked `ATOMIC` equal to its domain in one of the starting replicas
5. Propagate as many changes as possible without violating above rules



Final Algorithm

```
sync(O, A, B) =
  if A = B then (A,B)           -- equal replicas: done
  else if A = O then (B,B)      -- no change to A: propagate B
  else if B = O then (A,A)      -- no change to B: propagate A
  else if A = MISSING then (A,B) -- delete/modify conflict
  else if B = MISSING then (A,B) -- delete/modify conflict
  else if ATOMIC in (dom(A) U dom(B))
    and dom(A) <> dom(O)
    and dom(B) <> dom(O)
    and dom(A) <> dom(B)
    then (A,B)                   -- atomicity conflict
  else                           -- proceed recursively
    for each child k, let
      (Ak,Bk) = sync(O(k), A(k), B(k)) in
    let A' = { k -> Ak } in
    let B' = { k -> Bk } in
    (A',B')
```

Aside: Synchronizing Lists

Applying this algorithm to lists encoded as trees yields **element-wise** synchronization.

Tree Transformation Language

Lenses, Informally

A **lens** l consists of two functions:

- The **get** function, $l \nearrow$, maps a concrete tree to an abstract tree
- The **put** function, $l \searrow$, maps an (updated) abstract tree and an (original) concrete tree to a (updated) concrete tree

Example: Get

Concrete tree

$$\left\{ \begin{array}{l} \text{Pat} \mapsto \left\{ \begin{array}{l} \text{Phone} \mapsto \{ 333-4444 \mapsto \{ \\ \text{URL} \mapsto \{ \text{http://pat.com} \mapsto \{ \end{array} \right. \\ \\ \text{Chris} \mapsto \left\{ \begin{array}{l} \text{Phone} \mapsto \{ 888-9999 \mapsto \{ \\ \text{URL} \mapsto \{ \text{http://chris.org} \mapsto \{ \end{array} \right. \end{array} \right.$$

yields abstract tree:

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

Example: Put

New abstract tree

$$\left\{ \begin{array}{l} \text{Pat} \mapsto \{ 333-4321 \mapsto \{ \\ \text{Jo} \mapsto \{ 555-6666 \mapsto \{ \end{array} \right.$$

(plus original concrete tree) yields new concrete tree:

$$\left\{ \begin{array}{l} \text{Pat} \mapsto \left\{ \begin{array}{l} \text{Phone} \mapsto \{ 333-4321 \mapsto \{ \\ \text{URL} \mapsto \{ \text{http://pat.com} \mapsto \{ \end{array} \right. \\ \text{Jo} \mapsto \left\{ \begin{array}{l} \text{Phone} \mapsto \{ 555-6666 \mapsto \{ \\ \text{URL} \mapsto \{ \text{http://google.com} \mapsto \{ \end{array} \right. \end{array} \right.$$



Lenses, Formally

Write T for the set of trees.

A **lens** l is a pair of partial functions

- $l \nearrow$ from T to T (get)
- $l \searrow$ from $T \times T$ to T (put)

satisfying two laws:

- **GETPUT:** $l \searrow (l \nearrow c, c) = c$ if $c \in \text{dom}(l \nearrow)$
- **PUTGET:** $l \nearrow l \searrow (a, c) = a$ if $(a, c) \in \text{dom}(l \searrow)$

A Programming Language for Lenses

Question: How do we make it easy for (power) users to write lenses? How do we check that they are well formed?

Answer: By providing a domain-specific language in which all expressions denote well-formed lenses.

Some Primitive Lenses

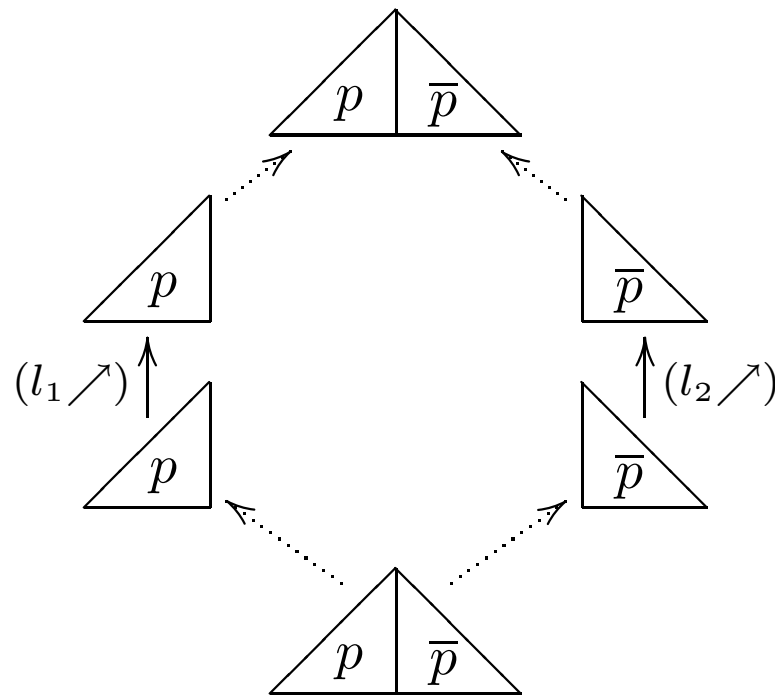
(In the get direction...)

- `id` does nothing
- `const v` transforms any concrete tree into v
- `l;k` applies l , then k
- `rename b` renames immediate children according to bijection b
- `hoist n` hoists the child under name n (which must be the only child)
- `map l` applies l to all immediate children



A Fun Primitive Lens: fork

The get direction of fork p l_1 l_2 :



Recursion

It is not hard to show that the collection of well-formed lenses forms a complete partial order.

Thus, our lens programming language may sensibly include definition by recursion.

Some Derived Lenses

- $\text{filter } p = \text{fork } p \text{ id } (\text{const } \{\})$
- $\text{focus } n = \text{filter } \{n\}; \text{hoist } n$
- $\text{mapp } p \ l = \text{fork } p \ (\text{map } l) \ \text{id}$
- $\text{hd} = \text{focus } *h$
- $\text{tl} = \text{focus } *t$
- $\text{map_list } l = \text{mapp } \{*h\} \ l; \text{mapp } \{*t\} \ (\text{map_list } l)$



Example: Bookmarks (Abstract)

```
{name -> Bookmarks Folder
  contents ->
    [{link -> {name -> Google
              url -> www.google.com}}
     {folder ->
       {name -> Conferences Folder
        contents ->
          [{link ->
            {name -> POPL
             url -> cristal.inria.fr/POPL2004}}]}}]}
```

Example: Bookmark lens

```
link = rename {dt = link};
  map (hoist *contents;
    hd {};
    hoist a;
    rename {href = url, *contents = name};
    prune add_date {$today};
    mapp {name} (hd {}; hoist PCDATA))
```

```
folder = rename {dd = folder};
  map (hoist *contents; folder_contents)
```

```
folder_contents =
  hoist_list [{h3} {dl}];
  rename {h3 = name, dl = contents};
  mapp {name} (hoist *contents; hd {}; hoist PCDATA);
  mapp {contents} (hoist *contents; map_list item)
```

Example: Bookmarks

```
item =
  dispatch [(dd), folder, folder)
            (dt), link, link)]

bookmarks =
  hoist *contents; hd {}; hoist html; hoist *contents;
  t1 {head -> {*contents -> [title -> {*contents ->
    [PCDATA -> Bookmarks]]}}}}};
  hd {}; hoist body; hoist *contents;
  folder_contents
```



Applications

Demos — Running

- universal bookmark synchronizer (Internet Explorer, Mozilla, Safari, ...)
- synchronizer for calendars in several formats
- (simple) structured text file synchronizer
- bibtex synchronizer

Demos — Planned

- address books, preference files, etc., etc., etc.
- richer structured documents (Word, LaTeX, DocBook, etc.)
- biological database annotations
- slide presentations (Powerpoint, Keynote)

Related Work

Related Projects

- IceCube
 - ongoing project at MSR (Shapiro, Rowstrom, Kemmarek, ...)
 - operation-based synchronization middleware
 - sophisticated algorithms for finding “best” merges of operation sequences from different replicas
- Bayou
 - late '90s project at Xerox PARC (Edwards, Mynatt, Petersen, Spreitzer, Terry, Theimer, ...)
 - operation-based
 - not as flexible as IceCube, but addressed distribution / scale issues very seriously
- Unison
 - late '90s project at Penn (Jim, Pierce, Vouillon)
 - state-based file synchronizer
 - formal specification similar to Harmony's

Related work

- Ramsey and Csirmaz
 - Careful algebraic specification of a file synchronizer similar to Unison, in an operation-based style
- LibreSource
 - current project at INRIA Lorraine [Molli, Oster, Skaf-Molli, Imine, etc.]
 - basic idea: **operation transform**

Define, for each pair of operations, op_1 and op_2 , a transformed version of op_1 , written $T(op_1, op_2)$, that achieves “the same effect” as op_1 but makes sense in a context where op_2 has been executed.

Finishing Up...

Harmony Status

- Core implementation and several demos running
- 2 users :-)

Ongoing Work

- tree-transformation language
 - additional primitives
 - binding
 - copying
 - characterization of expressive power
 - empirically (by building demos)
 - analytically
 - metatheory (type systems, algebraic theory, ...)
 - pushing the language further (e.g., database joins!)
 - generating lenses “by example” or from schemas
- multi-replica synchronization
- beyond tree-structured data
 - synchronizing dags [with Sanjeev Khanna and Alan Schmitt]
 - relations, ordered lists, sets, bags, etc., etc.



Want to play?...

Papers on our earlier synchronizer, Unison (as well as full sources, docs, and pre-built binaries) can be found here:

<http://www.cis.upenn.edu/~bcpierce/unison>

A paper on Harmony's lens language can be found here:

<http://www.cis.upenn.edu/~bcpierce/harmony>

More Harmony papers will appear here in due course.