harmony: a generic synchronization framework for heterogeneous, replicated data

Nate Foster
with Aaron Bohannon, Malo Denielou,
Michael Greenwald, Christian Kirkegaard,
Stéphane Lescuyer, Benjamin Pierce, and Alan Schmitt

Penn
University of Pennsylvania

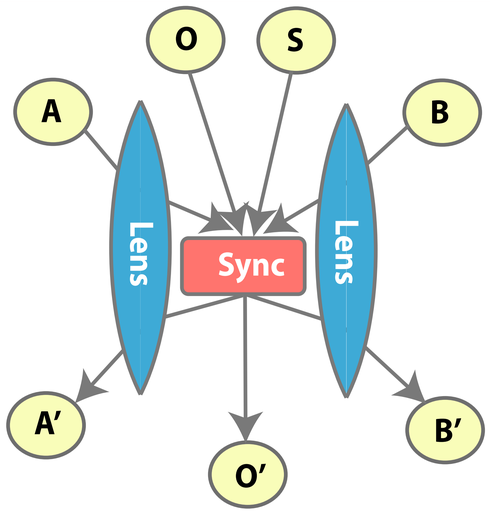
SUMMARY

Harmony is a framework that can be instantiated to build synchronizers for a wide variety of heterogeneous, tree-structured data. Building on the system's core implementation we have assembled synchronizers for address books, browser bookmarks, calendars, structured text, and several others. Harmony's design emphasizes the use of lenses to transform the replicas before (and after) synchronization. Lenses facilititate synchronization of heterogeneous data and also simplify the synchronizer because the replicas are pre-processed (and generally made less complicated). The synchronization algorithm is simple: it traverses the replicas, merging non-conflicting updates and issuing a local schema test. This local check is enough to ensure that the results are globally well-formed.

ARCHITECTURE

Harmony's architecture has two key components:

- **lenses:** view update translators for trees; used to map between heterogeneous concrete replicas and trees belonging to a common abstract synchronization schema;
- **synchronization algorithm:** a generic, local, state-based, schema-aware algorithm that merges the non-conflicting updates made to each replica.



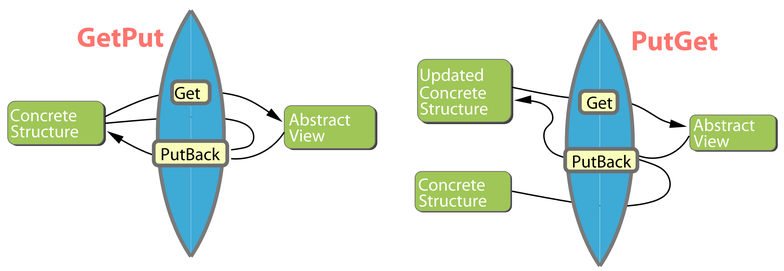
Wiring these components together the architecture is as depicted. A and B are the repliacs, the archive O represents the last synchronized state, S is the synchronization schema, A′ and B′ are updated replicas, and O′ an updated archive.

LENSES, SEMANTICALLY

A **"well-behaved" lens** is a pair of functions, Get and PutBack, with types:

- Get: $C \longrightarrow A$
- PutBack: $A \times C \longrightarrow C$

that obey two behavioral "round-trip" laws:



SYNCHRONIZATION ALGORITHM

```
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 = χ then (o,a,b)                     --unresolved conflict
 else if a = ⊥ and b ⊏ o then (a,a,a)            --a deleted more than b
 else if a = ⊥and b ⋢ o then (χ,a,b)             --delete/create conflict
 else if b = ⊥and a ⊏ o then (a,a,a)             --b deleted more than a
 else if b = ⊥and a ⋢ o then (χ,a,b)             --delete/create conflict
 else                                           --proceed recursively
   let (o'(k),a'(k),b'(k) = sync(S(k),o(k),a(k),b(k))
       ∀k∈ dom(a), dom(b) in
    if (dom(a') ∉ doms(S) or dom(b') ∉ doms(S))
      then (X,a,b)                              -- schema domain conflict
      else (o',a',b')
```

Definition: S is path consistent iff whenever t,u∈S we can update t along any path present in u and the resulting tree also belongs to the schema.
Theorem: if a, b∈S and S is path consistent, then a′, b′∈S and are the most synchronized safe results.

LENSES, SYNTACTICALLY

Focal is a language of combinators where every well-typed expression denotes a well-behaved lens. Every Focal program can be run forwards (get) and backwards (putback) with meaningful results in both directions. These are just a few of the interesting lenses in Focal:

- **id**: the identity;
- **compose**: puts two lenses in sequence;
- **const**: the constant lens;
- **hoist/plunge**: adds or removes an edge near the root;
- **fork/xfork**: splits the tree in two and applies a different lens to each part;
- **map/wmap**: applies a lens below every child;
- **acond/ccond**: conditionally selects a lens to apply;
- **μ**: defines recursive lenses.

ABSTRACT BOOKMARK SCHEMA

Browser bookmark data comes in many formats (e.g., Mozilla Firefox uses HTML, Safari XML, and IE the filesystem). The lenses in our bookmark synchronizer map each of these formats to trees belonging to a common abstract schema:

```
schema Link = {"name"=Value, "url"=Value}
schema Folder={"name"=Value, "contents"=Contens}
and Contents=List.T (Folder | {"link"=Link})
schema Abstract={"bookmark"=Contents, "toolbar"=Contents}
```

MOZILLA FIREFOX LENS

```
module Mozilla =

(* item: process a separator, folder, or link *)
let item : lens =
   acond {"HR" = Any} {"separator"={}}
       (const { "separator" = {} } {"HR" = {Xml.Children = []}})
       (schema CFolder = [{"H3" = Any },{"DL" = Any}] in
        hoist "DT";
        hoist Xml.CHILDREN;
        acond CFolder Bookmarks.Folder
            (protect (folder))
            (protect (link)))

(* folder: process a folder, recursively map item on contents *)
and folder : lens =
   hoist_nonunique List.HD { "H3" };
   rename "H3" "name";
   hoist_nonunique List.TL { `List.HD, `List.TL };
   xfork { `List.HD, `List.TL } { "contents" }
       (focus List.HD { `List.TL=[] };
        rename "DL" "contents")
       id;
   wmap { "name" ->  focus Xml.CHILDREN {};
                     List.hd [];
                     hoist Xml.PCDATA,
          "contents" -> hoist Xml.CHILDREN;
                        List.map item }

(* link: process a link, just a projection *)
and link : lens =
   List.hd [];
   rename "A" "link";
   wmap { "link" ->
       (xfork { `Xml.CHILDREN, "HREF" } { "name", "url" }
           (rename "HREF" "url";
            fork {"url"}
                id
                (focus Xml.CHILDREN {};
                 List.hd [];
                 rename Xml.PCDATA "name"))
       (const {} {})) }

(* l1: strips away prelude, apply item, fix up toolbar *)
let l1 : lens =
   List.hd [];
   hoist "DT";
   hoist Xml.CHILDREN;
   List.tl { "H1" = { `Xml.CHILDREN =
                          [{`Xml.PCDATA = {Bookmarks}}]}};
   List.hd [];
   hoist "DL";
   hoist Xml.CHILDREN;
   List.map item;
   xfork { `List.HD } { "toolbar" }
       (hoist List.HD;
        focus "contents" { "name" = { "BookmarksBar" } };
        plunge "toolbar")
       (rename List.TL "bookmarks")

(* l2: project away separators *)
let l2 : lens =
   let filter_seps : lens =
       List.filter
           ({"contents" = Any, "name" = Any} | { "link" = Any })
           { "separator" } {};
       List.map (wmap { "contents" -> filter_seps }) in
   l1;
   map filter_seps
```