# Exploring Filesystem Synchronization with Lightweight Modeling and Analysis

by

Tina Ann Nolte

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2002

Author . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
August 9, 2002

Certified by . . . . . . . . . . . . . . . . . . . . . . . .
Daniel Jackson
Associate Professor
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

# Exploring Filesystem Synchronization with Lightweight Modeling and Analysis

by

Tina Ann Nolte

Submitted to the Department of Electrical Engineering and Computer Science on August 9, 2002, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering

## Abstract

There are a number of software tools that offer a user the ability to synchronize filesystems despite conflicting updates made to multiple filesystem replicas, and often it is unclear what policies these tools employ and difficult to understand what the policies guarantee. Alloy, a lightweight formal modeling language, is used in an exploration of filesystem synchronization properties, specifications, and algorithms. The Alloy language and tool are evaluated in the task of interactively constructing models of filesystem synchronization and understanding their intricacies.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor

# Acknowledgments

I would like to thank J, my family, and most importantly, Kirby.

# Contents

# Chapter 1

# Introduction

As the world has become more mobile, replicated data has become more common. Unlike distributed filesystems that try to give all users the impression of simultaneously working off one shared filesystem, filesystem replicas in the mobile computing context must allow for disconnected operation by several users. Users synchronize data on their mobile machine with a desktop machine or with other users who may have updated the original data as well.

There are a number of software tools that offer a user the ability to synchronize data in the face of these issues, but often it is unclear what policies these tools employ and what the policies guarantee. These are important considerations since users do not want to discover that after synchronizing their mobile computer's data with another user's mobile computer they have lost whole files that may have been the product of hours of work. More subtly, users don't want to look at the synchronized filesystem and see that the directory structure and directory contents are as expected and continue with the belief that files have the changes they made to them only to find that some files have been replaced with a file of the same name from another user's filesystem.

## 1.1 Previous Work

Several commercial synchronizers exist, but their specifications (as described in user manuals and documentation) are often confusing and unclear. For example, here is one of the more detailed descriptions of the behaviour of Microsoft Briefcase available from the Microsoft website [9]:

> When you synchronize files, the files that you opened or updated while disconnected from the network are compared to the versions of the files

> that are saved on the network. As long as the same files you changed haven't been changed by someone else while you were offline, your changes are copied to the network.
>
> If someone else made changes to the same network file that you updated offline, you are given a choice of keeping your version, keeping the one on the network, or keeping both. To save both versions of the file, give your version a different file name, and both files will appear in both locations.
>
> If you delete a network file on your computer while working offline but someone else on the network makes changes to that file, the file is deleted from your computer but not from the network.
>
> If you change a network file while working offline but someone else on the network deletes that file, you can choose to save your version onto the network or delete it from your computer.
>
> If you are disconnected from the network when a new file is added to a shared network folder that you have made available offline, that new file will be added to your computer when you reconnect and synchronize.

This case by case description of the Briefcase specification is, in places, contradictory and incomplete (see Section 4.2). For example, it is not explicitly stated that if you do not change a file and somebody else does that you will ever receive an updated version of the file. Informal descriptions often suffer from these kinds of problems.

### 1.1.1 Formalizing Filesystem Synchronization

Work formalizing notions of filesystem synchronization has been done by Pierce and others [2, 1, 10, 11]. These papers have concentrated on establishing a rigorous framework for a discussion of synchronization, ranging from basic specfication studies to algebraic descriptions of filesystems. Pierce and Vouillon have implemented the Unison synchronizer based on the specifications they developed and proved correctness of a large part of the system with a theorem prover.

A different approach to studying a problem, as has been demonstrated by case studies in the past [8, 7], is to use an automated checking tool to experiment with specification and verify its correctness. The construction of a model of a system makes the task of experimenting with changes to policy or new algorithms much easier.

The Alloy language is well suited for the task of modeling synchronizers. The lightweight modeling aspects of the relational Alloy language [5, 6] are conducive to speedy expression of ideas, and the interactive nature of its tool gives one the power to quickly determine what is problematic in new policies or properties without having to implement the system or spend a great deal of time reasoning about all details of what can go wrong. Its simplicity as a language as well as the automatic capabilities of its tool are invaluable. Exhaustive checking of statements in a defined scope often produces "pathological" results, involving cases that manual test case generation would fail to consider. Feedback from the tool in the form of counterexamples combined with the ease of language expression make required corrections to a model easier to make. Alloy's structuring mechanisms are designed to describe systems with the complexity of data that a filesystem and synchronizers would have, which traditional model checkers (such as SPIN [4]) handle less effectively.

Pierce and Vouillon, in [10], have formalized a reference implementation of their implemented synchronizer, Unison, in the Coq proof assistant and verified properties of their specification. Theorem provers [3] such as Coq have the ability to prove that an algorithm is correct, which Alloy cannot do. However, theorem provers in general require a great deal of expertise to operate and when a verification task is not successful it can be difficult to determine if the wrong proof strategy was employed or, if it is a problem in specification, where that specification has failed. Alloy requires very little in the way of experience to get working and the counterexample generation is invaluable for pinpointing errors in specification.

## 1.2 Overview of Filesystem Synchronization

Filesystem synchronization can be split into two different tasks: *update detection* and *reconciliation*. Update detection is the identification of changes (updates) made to different replicas since the last synchronization. Reconciliation uses this information to produce new filesystems that take into account conflicting updates and reflect the updates made to the filesystems that are non-conflicting.

Update detection for any replica identifies changes made to the original filesystem. The reported possible updates at the least is *safe* (includes anything that has changed); it may include more changes than occurred. There is a spectrum of implementations of update detectors:

• Trivial detector: report everything as dirty (changed). This will, in general, produce a large number of spurious conflicts, but is a safe method by which to report changes to a filesystem; no user's new data is overwritten with data of another user.

• Exact detector: report only those things that have changed. While the reconciliation task benefits from the lack of spurious information, update detection will be more resource intensive.

• Variety of other policies. Many that may be used in real filesystems have subtle problems and are not safe, resulting in possible data loss. One example of this is the "last modified time" strategy used by several synchronizers found packaged with some operating systems. This strategy in general misses some updates, allowing files to be accidentally deleted. The strategy can be made safe by marking all files as dirty that have any ancestor that has been modified, though this will result in large portions of a filesystem being marked dirty.

Reconciliation is the second major component of synchronization. Given an original filesystem $O$, two filesystems $A$ and $B$ that are both updated replicas of $O$, and information on which portions of $A$ and $B$ are dirty, how do you produce new filesystems $A'$ and $B'$ so that $A'$ is the updated version of $A$ that incorporates the updates of $B$ that don't conflict with any updates in $A$ (and similarly for $B'$)? In practice more intelligent policies that make decisions based on user priority or file metadata can be adopted, but this is one of the most conservative (and arguably simplest) kinds of reconciliation that one might expect.

## 1.3 Contributions

Using the Alloy analyzer, I modeled and checked basic properties of synchronization and checked that some popular synchronizers do (and some don't) satisfy basic filesystem synchronization specifications. This work also highlighted the need for some changes to the Alloy language and tool.

Chapter 2 outlines one model of the filesystem synchronization task. Chapter 3 discusses the use of Alloy to model a detailed reconciliation algorithm and issues constructing models in the Alloy language. Chapter 4 describes the use of Alloy to model high-level specifications of filesystem synchronization, as well as interesting problems with specifications derived from informal descriptions of synchronization. Chapter 5 summarizes the lessons I have learned about some filesystem synchronization policies and observations about the difficulties and usefulness of the Alloy language. Appendix A gives a detailed tutorial description of the process of modeling a filesystem synchronization algorithm and a discussion of many of the problems that can occur. Complete models used in this document are included in Appendix B.

# Chapter 2

# Pierce and Balasubramaniam's Model

Here I will summarize the formalization of filesystem synchronization developed by Pierce and Balasubramaniam [2]. An abstract model of filesystems is presented and two phases of synchronization are identified: *update detection* and *reconciliation.*

## 2.1 Filesystems

Filesystems are finite partial functions from paths to either files or other filesystems. Paths are sequences of names. There is an empty path, $\varepsilon$. Paths $p$ and $q$ can be concatenated using the . operator (as in $p.q$). The mapping from paths to filesystems or files is constrained to guarantee that we consider mappings preserving standard tree-like filesystem structure: for any paths $p$ and $q$, following $p.q$ in a filesystem $F$ is the same as first following $p$ in $F$ and then following $q$ from the resulting subfilesystem.

## 2.2 Synchronization

Filesystem synchronization can be considered as two tasks: *update detection* and *reconciliation*[2, 1]. Here, we consider two replicas $A$ and $B$ being simultaneously synchronized against each other with respect to the original version $O$ of a filesystem.

### 2.2.1 Update Detection

Update detection computes a predicate *dirty* on paths such that if a path is not dirty (updated) in a replica $A$, then it maps to the same thing in the replica as it does in the original filesystem $O$:

$\forall p : Path \mid \neg dirty_A(p) => O(p) = A(p)$

For convenience's sake, dirtiness is up-closed; if path $p$ is a prefix of path $q$ and $q$ is dirty, then $p$ is as well:

$\forall p, q : Path \mid (p \leq q \; \&\& \; dirty_A(q)) \Rightarrow dirty_A(p)$

This makes defining reconciliation easier.

### 2.2.2 Reconciliation

Reconciliation is the second task of synchronization. We want reconciliation to propagate nonconflicting updates while preserving any user's local changes. What is a conflicting update? Given replicas $A$ and $B$, it is a portion of the filesystem that has been updated in both $A$ and $B$ to two distinct values different from the original. Given an original filesystem $O$, replicas of $O$ called $A$ and $B$, and reconciled versions of $A$ and $B$ called $A'$ and $B'$, for every path $p$ we expect the following to be true:

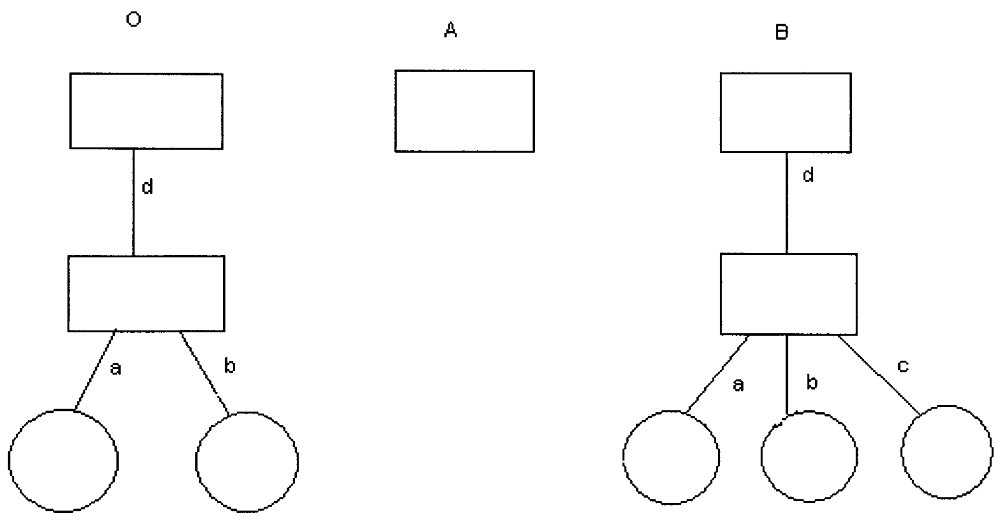
1) If $p$ is not dirty in $A$, then we know by up-closedness that the entire subtree rooted at $p$ in $A$ is not dirty. As a result, any updates from the corresponding subtree in $B$ should be propagated, giving $A'(p) = B'(p) = B(p)$.

2) Similarly if $p$ is not dirty in $B$.

3) If path $p$ is a directory in $A$ and $B$ then it is also a directory in $A'$ and $B'$.

4) If $p$ is dirty in $A$ and $B$ and is not a directory in both then we leave things as they are: $A'(p) = A(p)$ and $B'(p) = B(p)$.

However, these rules are not consistent. Consider this example where boxes indicate directories and circles indicate files:



Here there is an inconsistency in the specification since according to rule 1, because path $d.c$ is not dirty in $A$ we should have $A'(d.c) = B(d.c) = h$. However, by rule 4, since path $d$ is dirty in $A$ and $B$ but is not a directory in both then we should have $A'(d) = A(d)$, namely nothing.

As a result, we guarantee the four properties above only for paths $p$ where all proper prefixes of $p$ refer to directories in both $A$ and $B$ (called relevant paths).

Here is a general algorithm taken from [1] for calculating the synchronization $A'$ and $B'$ from $A$ and $B$ after a relevant path $p$. $isdir_{A,B}(p)$ is true when path $p$ indicates a directory in both filesystems $A$ and $B$. $children_{A,B}(p)$ is the set of immediate children of a path $p$ in either filesystem $A$ or $B$. The recursive portion of the algorithm is in step 2. If a directory is to be synchronized then each child path is synchronized one at a time, the result of which is passed into the synchronization of the next child path:

```
$recon(A, B, p) =$
1) if $\neg dirty_A(p) \bigwedge \neg dirty_B(p)$ then (A, B)
2) else if $isdir_{A,B}(p)$
   then let $\{p_1, p_2, \ldots, p_n\} = children_{A,B}(p)$
      in let $(A_0, B_0) = (A, B)$
         let $(A_{i+1}, B_{i+1}) = recon(A_i, B_i, p_{i+1})$ for $0 \leq i < n$
         in $(A_n, B_n)$
3) else if $\neg dirty_A(p)$ then $(filesystemOverwrite(A, B, p), B)$
4) else if $\neg dirty_B(p)$ then $(A, filesystemOverwrite(B, A, p))$
5) else $(A, B)$
```

where $filesystemOverwrite(T, S, p)$ is the result of replacing the subtree rooted at $p$ in $T$ with that of $S$ at $p$.

## 2.3 Properties of Synchronization and Reconciliation

To discuss properties of synchronization and reconciliation it is helpful to formally define the notion of synchronizations.

*AfterPSynchronization* captures the notion that filesystems $A'$ and $B'$ are synchronizations of $A$ and $B$ for all paths with $p$ as their prefix. In more detail, $A'$ and $B'$ are an *afterPSynchronization* of $A$ and $B$ from $p$ if the following is true for all relevant paths $p.q$ in $A$ and $B$:

1. $\neg dirty_A(p.q)$
   $\Rightarrow A'(p.q) = B'(p.q) = B(p.q)$
2. $\neg dirty_B(p.q)$
   $\Rightarrow A'(p.q) = B'(p.q) = A(p.q)$
3. $isdir_{A,B}(p.q)$
   $\Rightarrow isdir_{A',B'}(p.q)$
4. $dirty_A(p.q) \bigwedge dirty_B(p.q) \bigwedge \neg isdir_{A,B}(p.q)$
   $\Rightarrow A'(p.q) = A(p.q) \bigwedge B'(p.q) = B(p.q)$

This aids in defining a synchronization of two filesystems after a path $p$ such that nothing else changes. We'll call this $sync_p(A', B', A, B)$:

1. $(A', B')$ is an $afterPSynchronization$ of $(A, B)$ after $p$
2. for all paths $q$, if neither $p$ nor $q$ is a prefix of the other
   then $A'(q) = A(q)$ and $B'(q) = B(q)$
3. if $q$ is a prefix of $p$ and $isdir_{A,B}(q)$ then $isdir_{A',B'}(q)$

There are two main properties one would like to be able to check about synchronization and reconciliation [2, 1, 10]:

• Synchronizations are fully characterized by the description above; for any relevant path p in updated filesystems $A$ and $B$ and dirtiness predicates $dirty_A$, and $dirty_B$, if $(A', B')$ and $(A'', B'')$ are both *afterPSynchronizations* of $(A, B)$ then $A' = A''$ and $B' = B''$.

• The reconciliation algorithm described actually satisfies properties of synchronization; if $recon(A, B, p) = (A', B')$ then $sync_p(A', B', A, B)$.

# Chapter 3

# Modeling Filesystem Synchronization in Alloy

Here the models of filesystem synchronization are written in the Alloy language. I will first describe some basic Alloy and then present a model of filesystem synchronization in that language as well as some properties that were checked with the Alloy tool.

## 3.1 Basic Alloy

*Types* Alloy assumes a universe of atoms partitioned into subsets corresponding to basic types. Mathematical relations (collections of tuples of atoms) are used to introduce the only composite datatypes. All relations are first order, meaning that elements of a tuple are atoms and not relations themselves. Expressions' values are always relations.

Sets are unary relations. Scalars are unary, singleton relations. No distinction is made between an atom, a set containing that atom, or a tuple of the atom. This leads to a more uniform syntax and simplifies the semantics of the language.

*Expression Operators* Standard set operators written as ASCII operators can be used to form expressions: + (union), & (intersection), and - (difference). The dot operator is generalized relational composition [6]. When $e$ is a set, $e.r$ denotes the image of the set $e$ under the relation $r$. If $e$ is a binary relation the composition is relational composition. Cross product is represented by the arrow operator ($p$->$q$).

*Formula Operators* Elementary formulas are constructed from the subset operator *in*. $s$ *in* $t$ says that expression $s$ denotes a subset of the expression $t$. Equality (=) is short for subset constraint in both directions.

Multiplicity markings (+ for one or more, ! for exactly one and ? for 0 or 1) are used to constrain relations. *r:S m -> n T* where $m$ and $n$ are multiplicity symbols constrains $r$ to be a relation that maps each $S$ to $n$ atoms of $T$ and $m$ atoms of $S$ to each $T$.

Alloy also uses standard logical operators: && (and), || (or), ! (not), => (implies), and <=> (iff). An *if-then-else* construct is also present in the language as syntactic sugar. When written within curly braces ({}) formulas are implicitly conjoined.

Quantifications are written in their usual form. For example, *all x:e | F* is true when the formula $F$ holds for every element in a set $e$. The quantifiers *some* and *no* are available as well. Lastly, set comprehensions constructing tuples with elements $x_1 ... x_n$ taken from expressions $e_1 ... e_n$ that satisfy a formula $F$ are written as follows: $\{x_1 : e_1, x_2 : e_2, ... x_n : e_n \mid F\}$.

*Signatures* Signature declarations introduce basic types. The declaration *sig S {f: E}* declares a type $S$ and a field (relation) *f*. This relation has a type from $S$ to the type of *E*. Not all declarations introduce new basic types. The keyword *extends* can be used to indicate that the type being declared is a subset of the supersignature listed. For example, if there were:

```
sig Node {}
sig File extends Node {}
disj sig Filesystem extends Node {
  contents: Node
}
```

this would indicate that *File*s and *Filesystem*s are subsets of *Node*s. The *disj* keyword indicates disjoint subsignatures; in this example, any Nodes designated as Filesystems will not also be Files. Use of the *static* keyword before a signature declaration constrains the model to have only one atom corresponding to the signature.

An assignment of a collection of signature declarations is an assignment of values to signatures and fields. Take for example the following assignment for the specification above:

*Node* = {n0, n1},
*File* = {n0},
*Filesystem* ={n1},
*contents* = {(n1, n0)},

corresponding to a universe with 2 nodes, the first of which is a file and the second of which is a Filesystem. The *contents* relation just contains one tuple from filesystem n1 to node n0.

Facts constrain constants of the specification. For example,

```
        fact {File + Filesystem = Node}
```

constrains *Nodes* to be *Files* or *Filesystems*.

*Function Applications* Alloy also has a useful but sometimes subtle concept of functions. A function's meaning is a set of assignments that include bindings to parameters. Function application may be either as an expression or as a formula. If a formula, the application is short for the function body where parameters are appropriately replaced by expressions from the application. For example:

```
        fun isDirAB (f, g: Filesystem, p: Path) {
          isDir(f, p) && isDir(g, p)
        }
        fact {all a, b: Filesystem, q: Path |
          (isDir(a, q) && isDir(b, q)) <=> isDirAB(a, b, q)
        }
```

The function *isDirAB* is used as a formula in the fact. If the application is as an expression, things get more interesting. You can introduce a result argument with type $C$ with this shorthand:

```
        fun f(a:A, b:B): C {...}
```

The keyword *result* is used in the function body to refer to the anonymous result of the function. For example,

```
        fun dirty(f, g: Filesystem): set Path {
          all p: Path | {
            p !in result => p.f::contents = p.g::contents
          }
        }
```

"returns" a set of Paths that can be substituted in for *result* in the body of the function while keeping the function true.

Functions with results can also be used as simple formulas with boolean values by passing a possible "return" value for checking in as the second argument. For example, the *dirty* function described above can be used to check if a set of paths $P$ could be substituted for the *result* keyword in the *dirty* function:

```
    dirty(f, P, g)
```

*Assertions* Assertions are formulas that claim to be valid. They do not constrain the atoms of the universe and are instead statements used in checking correctness of the model.

## 3.2 Alloy Model of a Filesystem Synchronizer

### 3.2.1 Filesystems

The first step to modeling filesystem synchronization is to model filesystems. Since synchronization deals with names, paths, and filesystem nodes it makes sense to introduce three different kinds of atoms: Names, Paths, and Nodes. Names will identify files and directories and paths will be sequences of names. Nodes are filesystem nodes (files and directories).

```
sig Name {}
sig Path {
  names: Seq[Name]
}
sig Node {}
```

All immediate pathprefixes of a path are also paths. This fact, since true of all paths, will give us that all pathprefixes of a path are paths:

```
fact NoMissingPathBegin {
  all p: Path | {
    !EmptyPath(p) => {
    some q : Path | (PathPrefix(q, p) &&
                    (#PathLength(p) = #PathLength(q) + 1))
    }
  }
}
```

We also need a restriction that all possible postfixes (tails) of a path are actually paths as well. Together with NoMissingPathBegin this gives us that all subpaths of a path are paths. This is necessary since if a path $p.q$ is mapped in a filesystem $A$, then path $p$ must exist and map to some filesystem $B$ and path $q$ must map from filesystem $B$ to some node:

```
fun tail (p: Path, q: Path) {
  p.names..SeqRest() = q.names
}
```

```
fact NoMissingPathEnd {
  all p: Path | {
    !EmptyPath(p) => {
            some q: Path | tail(p, q)
    }
  }
}
```

Nodes will be either files or filesystems.

```
disj sig File extends Node {}
disj sig Filesystem extends Node {}
fact FileDirPartition {
  File + Filesystem = Node
}
```

We add a contents relation to the Filesystem signature to model directory contents.

```
disj sig Filesystem extends Node {
  contents: Path ? -> ? Node
} {
  some p: Path | EmptyPath(p) && this = p.contents
}
fun EmptyPath (p: Path) {
  p.names..SeqIsEmpty()
}
```

Note the multiplicity constraints expressing that contents are a partial mapping from some paths to at most one node. Also, note the additional constraint ensuring that following the empty path from a node leads back to that node. Incidentally, the statement also ensures that there is an empty path. Empty paths are defined as paths where the sequence of names it represents is empty.

Next, since it is possible for two different paths to be comprised of the same sequence of names in this model, we want to canonicalize the atoms that are being considered.

```
fact CanonicalPath {all a, b: Path| a.names = b.names => a = b}
fact CanonicalSequence {all a, b: Seq[Name] | a..SeqEquals(b)
                                                    => a = b}
```

A similar issue exists with nodes. To canonicalize nodes, we first have to include a notion of equivalence for nodes expressing that two nodes are equivalent if they are the same node or if they are filesystems mapping the same paths such that if a path is mapped to a file in one then it is mapped to the same file in the other. Note the use of *result* in the *Paths* function. *Paths* "returns" a set of *Path*s, the value of which is the set of paths mapped by the filesystem's *contents* relation:

```
fun Paths(f: Filesystem): set Path {
  result = Node.~(f.contents)
}
fun EquivNode (f, g: Node) {
  (f = g) ||
  (f in Filesystem && g in Filesystem && Paths(f) = Paths(g) &&
  all p: Path | all e: File | {
    p->e in f.contents <=> p->e in g.contents
  })
}
fact CanonicalNode {all a, b: Node | EquivNode(a, b) => a = b}
```

We need to start considering constraints on the atoms to make sure that filesystems behave as we expect; for any paths $p$ and $q$, looking up a composite path $p.q$ is the same as first looking up the first portion of the path, $p$, and then looking up the remainder of the path, $q$, from the resulting sub-filesystem.

However, this constraint on filesystems would disallow existence of files in filesystem contents. Why? Consider a path $p$. Say that in a particular filesystem this path leads to a file. Following the empty path $\varepsilon$ from a file yields nothing since files don't have path contents. Since path $p$ can be expressed as $p.\varepsilon$, the filesystem in question would map path $p$ both to the file (since $p$ leads to a file) and to nothing (since following $p$ in the filesystem is the same as following $p$ to the file it maps to and then following $\varepsilon$ to nothing). As a result, this has to be treated as a special case to be considered in the description of the tree structure constraints (*):

```
    fact TreeStructure {
      all f: Filesystem | all p, q: Path | {
        let g = p.f::contents | {
(*)       (g in File && EmptyPath(q)) ||
              (q.g::contents = append(p, q).f::contents)
        }
      }
}
```

### 3.2.2 Update Detection

To model update detection, or dirtiness, it is convenient to add an additional signature with a field mapping an original node and a replica node to a set of paths that have been updated. This represents the notion of there being a piece of synchronization software making decisions about which paths are to be marked dirty:

```
static sig Synchronizer {
  dirty : Node -> Node -> Path
}
```

The next step is to constrain the dirty relation to be both safe and up-closed as discussed earlier. Remember, an update detection policy is safe if it finds all paths where a filesystem's contents have changed from the original and it is up-closed if prefixes of dirty paths are also dirty:

```
fact DirtySafe {
  all f, g: Filesystem | all p: Path | {
    p !in Synchronizer.dirty[f][g] => p.f::contents = p.g::contents
  }
}
fact DirtyUpClosed {
  all f, g: Filesystem | all p, q: Path | {
    (PathPrefix(p, q) && q in Synchronizer.dirty[f][g]) =>
                p in Synchronizer.dirty[f][g]
  }
}
```

Other notions of dirtiness can be assumed by changing the above two facts to constrain the dirtiness mapping appropriately.

### 3.2.3 Reconciliation

Reconciliation is arguably the most complicated aspect of synchronization. Recall the algorithm described in Chapter 2:

```
$recon(A, B, p) =$
$1)$ if $\neg dirty_A(p) \bigwedge \neg dirty_B(p)$ then (A, B)
$2)$ else if $isdir_{A,B}(p)$
    then let $\{p_1, p_2, \ldots, p_n\} = children_{A,B}(p)$
        in let $(A_0, B_0) = (A, B)$
            let $(A_{i+1}, B_{i+1}) = recon(A_i, B_i, p_{i+1})$ for $0 \leq i < n$
             in $(A_n, B_n)$
$3)$ else if $\neg dirty_A(p)$ then $(filesystemOverwrite(A, B, p), B)$
$4)$ else if $\neg dirty_B(p)$ then $(A, filesystemOverwrite(B, A, p))$
$5)$ else $(A, B)$
```

To model this reconciliation algorithm from [1], we start by introducing a reconciliation relation to the Synchronizer atom to represent the reconciliation of two nodes $A$ and $B$ from an original $O$ and a path $p$. *Synchronizer.recon*$[O][A][B][p]$ corresponds to $recon(A, B, p)$ given an original filesystem $O$ by which to calculate dirty paths:

```
static sig Synchronizer {
  recon: Node -> Node -> Node -> Path -> !(Node->Node),
  dirty : Node -> Node -> Path
}
```

The next step is constraining the *recon* field to properly describe the reconciliation function (Figure 3-1).

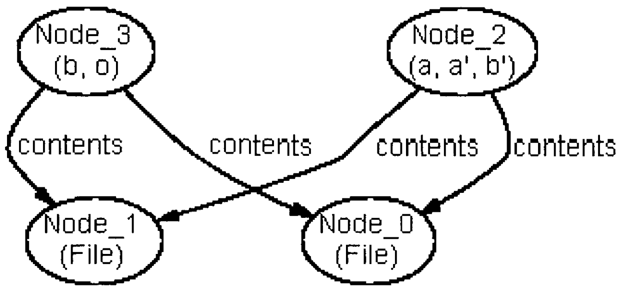
Lines 5 and 6 correspond to part 1 of the reconciliation algorithm. Lines 26-28, 29-31, and 32 correspond exactly to parts 3, 4, and 5 in the reconciliation algorithm. The only departure from the original algorithmic description is in the recursive portion (lines 7-25 above); an abstraction had to be considered where $a'$ and $b'$, the result of synchronizing a particular path $p$ in two filesystems, is related to the synchronization (the pair $c$ and $d$ on line 14 and 15) of each child on that path. Examining a child path $q$ of a synchronized path $p$ in two filesystems will reveal that $q$ in the final synchronized filesystems $a'$ and $b'$ maps to the same nodes as it would have if the replicas were synchronized at path $q$.

Now we can examine some simple examples of synchronization at work. The Alloy analyzer can automatically produce sample systems that satisfy constraints in the constructed model. The tool also provides a visualization utility that can be customized to display interesting portions of a model. The following example is produced with this visualization utility from a model automatically generated by Alloy. Note that in the example the filesystems utilize sharing of atoms; some filesystem nodes are used in more than one filesystem structure, such as in the case of Node 1

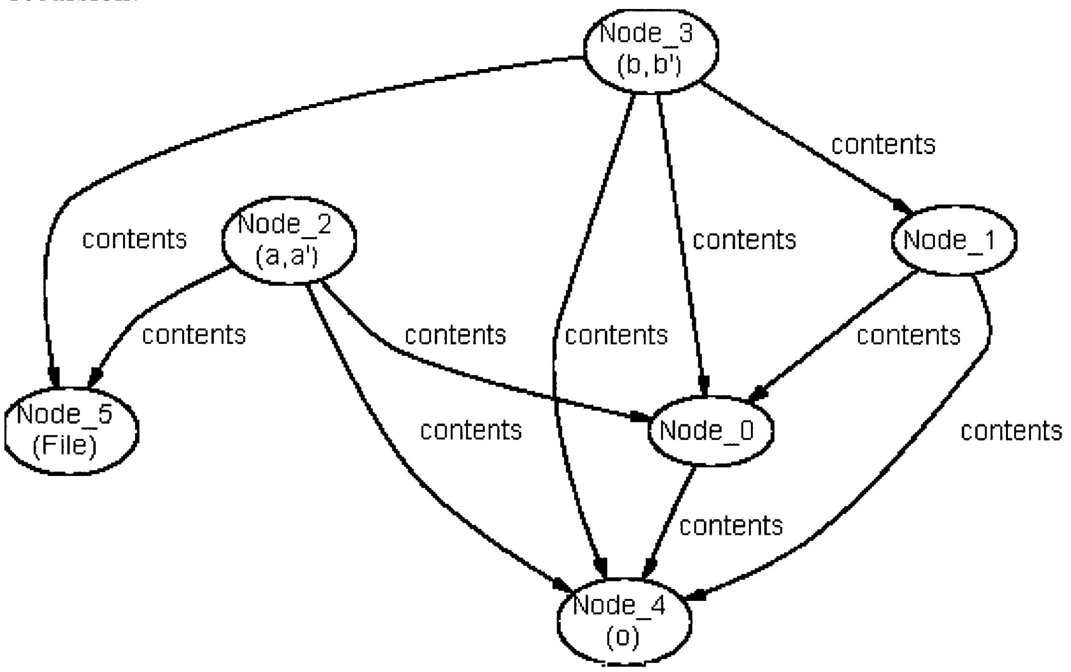
```
fact reconFacts {
  all o, a, b: Node | all p: Path | some abp, bap: Node |{
    let dirtya = Synchronizer.dirty[o][a], dirtyb =
                   Synchronizer.dirty[o][b] | {
      (p !in dirtya + dirtyb)
           => Synchronizer.recon[o][a][b][p] = a->b
      else isDirAB(a, b, p)
           => {
       no childrenAB(a, b, p)
           => Synchronizer.recon[o][a][b][p] = a->b
       all q: childrenAB(a, b, p) | {
         let a' = Node.~(Synchronizer::recon[o][a][b][p]),
             b' = Synchronizer.recon[o][a][b][p][a'],
             c = Node.~(Synchronizer.recon[o][a][b][q]),
             d = Synchronizer.recon[o][a][b][q][c] | {
              (children(a', p) + children(b', p)) in
                                              childrenAB(a, b, p)
               all s: Path | {PathPrefix(s, q) =>
                       ((s in Paths(a') <=> s in Paths(c)) &&
                        (s in Paths(b') <=> s in Paths(d))) }
               (q.a'::contents = q.c::contents && q.b'::contents =
                                                  q.d::contents)
               }
         }
       }
      else p !in dirtya
           => filesystemOverwrite(a, abp, b, p) &&
                  Synchronizer.recon[o][a][b][p] = abp->b
      else p !in dirtyb
           => filesystemOverwrite(b, bap, a, p) &&
                  Synchronizer.recon[o][a][b][p] = a->bap
      else Synchronizer.recon[o][a][b][p] = a->b
    }
  }
}
```

Figure 3-1: Alloy Model of the Reconciliation Algorithm

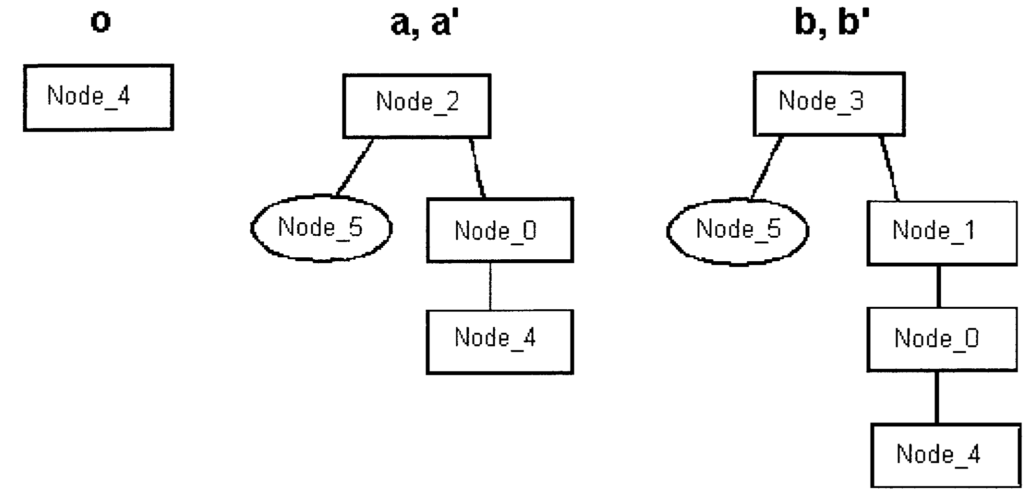
which is mapped to by both Node 3 and Node 2 as separate filesystems. The example of synchronization is one of the simplest where only one replica ($a$) differs from the original ($o$) and both filesystems in the resulting synchronization ($a'$ and $b'$) are the updated filesystem:



Here is a more complicated example of reconciliation from the empty path requiring recursion:



This picture is difficult to understand, though. First, atom sharing is present; some atoms such as Node 4 are mapped to by several filesystems even though there is no significance to their multiple occurrences. Also, the *contents* relation maps paths rather than names. As a result, there is an arrow drawn from a filesystem node to each of its descendants, whether an immediate child or not. To better understand what is going on in this example I translated it to a more standard representation of filesystem structure where boxes are filesystems and circles are files, shared atoms are copied between filesystem structures, and filesystems only point to immediate children:



The original filesystem ($o$) is an empty directory. The two replicas ($a$ and $b$) each have added directories. The *recon* algorithm does employ recursion in this case, but since all paths in both replicas are dirty, no changes are propagated ($a'$ and $b'$).

The complete model for this section is in Appendix B.3. To describe a different reconciliation algorithm, all that would have to be done is to change the reconFacts fact to describe the new algorithm.

## 3.3 Analyzing the Model

Since Alloy supports declarative specification, models can be incrementally developed. The Alloy analyzer then evaluates partially completed models for subtle properties by translating the model being analyzed into a Boolean formula, dispatching the formula to a SAT solver and finally translating any solutions found by the solver into an instance of the model.

Since Alloy is not decidable, the analyzer does not provide sound and complete analysis of models. However, it does check models using an exhaustive search in a given finite "scope" provided by the user. This scope defines the number of atoms present in each basic type. The Alloy analyzer then returns one of two things: a solution or a message indicating no solution was found. If an assertion is being tested, a solution is an instantiation of atoms constrained by the constructed model that provide a counterexample to the assertion. A solution when a function is being evaluated provides an instantiation of atoms that satisfies not only the model, but the function as well.

Since these searches are conducted in finite scope it is possible that even though no solutions are found in some scope a solution could be found in a larger scope. However, in practice, if counterexamples exist, one can often be found in small scope.

There are a number of properties that are important to verify about synchronization [2, 1, 10]. In this section we will first formalize some notions of synchronization to be used in analysis of the models and then check three properties:

- Uniqueness of Synchronization
- Soundness of Reconciliation
- Completeness of Reconciliation

### 3.3.1 Formalization of Synchronization Specification

In order to check any properties about synchronization, some concepts have to be formalized. One concept is that of a relevant path:

```
fun relevant (a, b: Filesystem, p: Path) {
  all q: Path | {
    PPathPrefix(q, p) => isDirAB(a, b, q)
  }
}
```

A relevant path is one where all proper pathprefixes correspond to directories in both filesystems being considered. This is how the problem with contradictory synchronization results was handled in the description of synchronization. *AfterPSynchronization* captures the notion that filesystems $A'$ and $B'$ are synchronizations of $A$ and $B$ starting from a particular path. Remember from Chapter 2 that $A'$ and $B'$ are an *afterPSynchronization* of $A$ and $B$ from $p$ if the following is true for all relevant paths $p.q$ in $A$ and $B$:

1. $\neg dirty_A(p.q)$
   $\Rightarrow A'(p.q) = B'(p.q) = B(p.q)$
2. $\neg dirty_B(p.q)$
   $\Rightarrow A'(p.q) = B'(p.q) = A(p.q)$
3. $isdir_{A,B}(p.q)$
   $\Rightarrow isdir_{A',B'}(p.q)$
4. $dirty_A(p.q) \bigwedge dirty_B(p.q) \bigwedge \neg isdir_{A,B}(p.q)$
   $\Rightarrow A'(p.q) = A(p.q) \bigwedge B'(p.q) = B(p.q)$

This is directly translatable into an Alloy function:

```
fun afterPSynchronization (a, b, a', b': Node,
                            dirtya, dirtyb: set Path, p: Path) {
  relevant(a, b, p)
  all pq: Path | {
    (PathPrefix(p, pq) && relevant(a, b, pq))
       => {
      pq !in dirtya => (pq.a'::contents = pq.b'::contents &&
                        pq.a'::contents = pq.b::contents)
      pq !in dirtyb => (pq.a'::contents = pq.b'::contents &&
                        pq.a'::contents = pq.a::contents)
      isDirAB(a, b, pq) => isDirAB(a', b', pq)
      (pq in dirtya && pq in dirtyb && !isDirAB(a, b, pq)) =>
                       (pq.a'::contents = pq.a::contents &&
                        pq.b'::contents = pq.b::contents)
      }
  }
}
```

This function is used by *syncp* which checks not only that *afterPSynchronization* holds but also that paths other than the synchronization path $p$ and any descendants remain the same. This is done by stating that for all paths $q$, if $q$ is not a pathprefix of $p$ and $p$ is not a pathprefix of $q$ then the synchronized replicas do not change their contents at $q$ because of synchronization. If $q$ is a pathprefix of the synchronization point and is a directory, then it remains so in the final synchronized version.

```
fun syncp (p: Path, a', b', a, b: Node, dirtya, dirtyb: set Path) {
  afterPSynchronization(a, b, a', b', dirtya, dirtyb, p)
  all q: Path | {
    incomparable(p, q) =>
       (q.a'::contents = q.a::contents &&
        q.b'::contents = q.b::contents)
    (PathPrefix(q, p) && isDirAB(a, b, q)) => isDirAB(a', b', q)
  }
}
```

### 3.3.2 Uniqueness of Synchronization

When synchronizing two replicas of a filesystem, users do not want to hear that non-deterministic decisions are being made about which portions of their filesystem are being preserved. Users want an assurance that the term "synchronization" refers to exactly one pair of filesystems. Synchronizations are "unique" if, given an original filesystem, two replicas, and a particular update detection policy, any pair of filesystems that is a valid synchronization is the same as any other pair that is a valid synchronization:

```
assert Uniqueness {
  all a, b, o: Filesystem | all dirtya, dirtyb: set Path | {
    all p: Path | all a1, b1, a2, b2: Filesystem | {
      (dirty(o, dirtya, a) && dirty(o, dirtyb, b) &&
       afterPSynchronization(a, b, a1, b1, dirtya, dirtyb, p) &&
       afterPSynchronization(a, b, a2, b2, dirtya, dirtyb, p)) =>
                          (p.a1::contents = p.a2::contents &&
                           p.b1::contents = p.b2::contents)
    }
} }
```

This assertion checks for scopes up to 6. This means that in the space of instantiations with 6 atoms per basic type there is no instantiation of the variables consistent with the model that does not satisfy the requirements of the assertion. A scope of 6 atoms is sufficient for the Alloy analyzer to generate cases involving conflicts and requiring recursion to resolve.

### 3.3.3 Soundness of Reconciliation

Users want to know that a piece of software they employ to synchronize their filesystem does not suffer from potential data loss. It is important to be certain that the reconciliation algorithm modeled in section 3.2.3 actually calculates a safe synchronization of the filesystems in question. Given two replicas $a$ and $b$ of an original filesystem $o$, we check that for all relevant paths $p$ in $a$ and $b$, if we reconcile $a$ and $b$ against $o$ starting at path $p$, the result is a synchronization of $a$ and $b$ from $p$:

```
assert Soundness {
  all a, b, o: Filesystem | {
    let dirtya = Synchronizer.dirty[o][a], dirtyb =
                                     Synchronizer.dirty[o][b]  | {
      all p: Path | all a', b': Filesystem | {
        (relevant(a, b, p) && Synchronizer.recon[o][a][b][p] = a'->b')
                          =>(syncp(p, a', b', a, b, dirtya, dirtyb))
      }
    }
  }
}
```

This assertion finds no counterexamples in small scope (4 atoms). The assertion is checkable for 6 atoms in a related model employing techniques to reduce memory usage. This lends some credibility to the hypothesis that the algorithm for reconciliation does calculate a synchronization of two filesystems against an original. Checking this model for 4 atoms requires about 5 minutes on a Pentium 4 processor with 256 MB of RAM, though there are ways to write this model more efficiently so that analyses require much less time.

### 3.3.4 Completeness of Reconciliation

Software is not particularly useful if it does not run on most inputs. Completeness of the reconciliation algorithm is concerned with whether or not reconciliation just bothers to synchronize some filesystems and not others. A natural way to express this would be to say that for any original filesystem $O$, and two updates $A$ and $B$, there is a pair of filesystems $A'$ and $B'$ that the reconciliation relation maps to. However, this has a counterexample in the Alloy model since there can be filesystems whose synchronizations don't fit in the scope being considered.

As a result, we instead check that if in the instantiated universe there exists a synchronization of two filesystems, the *recon* relation maps those filesystems:

```
assert reconGensAll {
  all o, a, b: Filesystem | all z: Path | {
    let dirtya = Synchronizer.dirty[o][a], dirtyb =
                          Synchronizer.dirty[o][b]  | {
        (some dirtya && some dirtyb && some a', b': Filesystem |
                        syncp(z, a', b', a, b, dirtya, dirtyb))
                          => some Synchronizer.recon[o][a][b][z]
        }
      }
    }
```

No counterexamples occur in a scope of 4 (though it is possible to examine the assertion in a scope of 6 in a more efficient modified version). This allows us to conclude with a degree of confidence that the model of reconcilation actually covers all cases of synchronization that it should.

This model can be found in its entirety in Appendix B.3. The alternative formulation of the model that is checkable in larger scope is found in Appendix B.2.

# Chapter 4

# Modeling Specifications of Implemented Synchronizers

Alloy's declarative statements allow for modeling systems at a much higher level than the algorithmic. Here we will examine some high-level specifications of filesystem synchronizers in the framework of synchronization defined in Chapter 3.

## 4.1 Unison Specification

As described in the Unison reference manual [12], update detection and reconciliation are of a very different flavor from the model presented in the previous sections. Update detection is described as finding changes to filesystem "contents" and no algorithm is provided for reconciliation; instead a description of the results of application of an algorithm are provided.

### 4.1.1 Update Detection and Conflicts

In order to model the Unison filesystem synchronizer specification, we need to get a handle on some notions used by the authors of the synchronizer. First, the contents of a path $p$ in a particular replica could be a file (string of bytes), a directory (designated by the token "DIRECTORY"), or absent (if $p$ does not refer to anything at all in that filesystem). A path is updated (in some replica) if its current contents (the string of bytes, "DIRECTORY" token, etc.) are different from its contents the last time it was successfully synchronized. A path is said to be *conflicting* if:

1. it has been updated in one replica,
2. it or any of its descendants has been updated in the other replica,
3. its contents in the two replicas are not identical.

With this description, conflicts are easy to model. *Samesort* is true when two nodes are the same or at least are both directories. It captures the notion of two nodes having the same contents:

```
fun samesort(a, b: Node) {
  a = b || (some (a & Filesystem) && some (b & Filesystem))
}
fun conflict(o, a, b: Node) {
  !(samesort(a, b) || o = a || o = b)
}
```

### 4.1.2 Reconciliation

Unison's reconciliation specification is described in its user manual as several tasks:

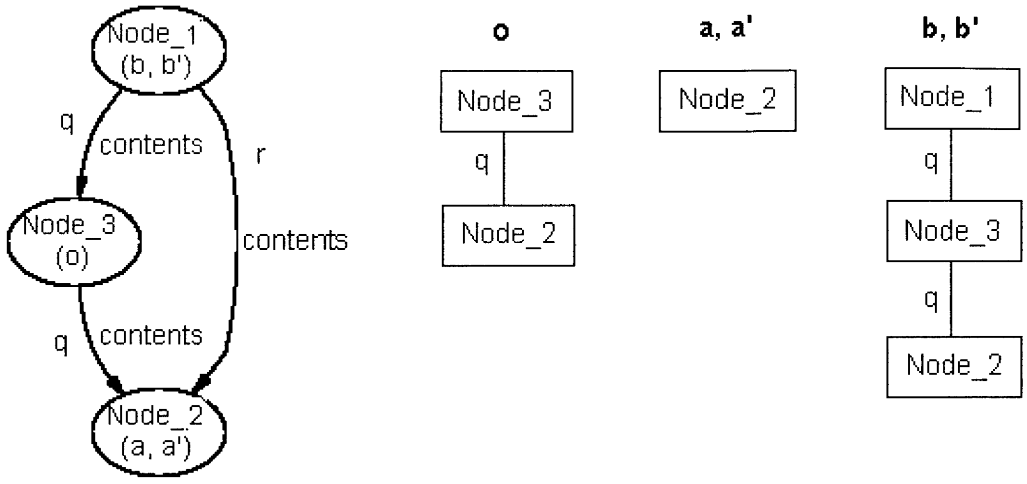
1. It checks for "false conflicts" — paths that have been updated on both replicas, but whose current values are identical. These paths are silently marked as synchronized in both replicas.
2. For updates that do not conflict, it propagates the new contents from the updated replica to the other.
3. Conflicting updates are left as is.

Here is a straightforward attempt at describing these properties. Parenthesized numbers correspond to lines from the description of the Unison specification above:

```
    fun BadUnisonRecon(o, a, b, a', b': Node) {
      all p: Path | {
(1)     (samesort(p.o::contents, p.a::contents) &&
                          samesort(p.o::contents, p.b::contents))
             => (samesort(p.a::contents, p.a'::contents) &&
                             samesort(p.b::contents, p.b'::contents))
(2)     else (samesort(p.o::contents, p.a::contents) &&
               !conflict(p.o::contents, p.a::contents, p.b::contents))
             => (samesort(p.b::contents, p.a'::contents) &&
                             samesort(p.b::contents, p.b'::contents))
(2)     else (samesort(p.o::contents, p.b::contents) &&
               !conflict(p.o::contents, p.a::contents, p.b::contents))
             => (samesort(p.a::contents, p.a'::contents) &&
                             samesort(p.a::contents, p.b'::contents))
(3)     else (samesort(p.a::contents, p.a'::contents) &&
                             samesort(p.b::contents, p.b'::contents))
```

```
        }
    }
```

However, this is not correct due to an inconsistency in the description of Unison's reconciliation specification given above. Here is a correct example of a synchronization that BadUnisonRecon rejects as a synchronization:



This example reveals a problem in the specification of BadUnisonRecon. Contradictory predictions about the value of $a'$ and $b'$ result in BadUnisonRecon not accepting any synchronization. There are three paths: an empty path, a path of length one ($q$), and a path of length two ($r$). On one hand, for path $q$, part (3) of BadUnisonRecon implies that:

•$q.a'$ :: *contents* $= q.a$ :: *contents* (nothing)

•$q.b'$ :: *contents* $= q.b$ :: *contents* (Node_3)

However, examine path $r$. In this case, part (2) of BadUnisonRecon applies, implying:

•$r.b'$ :: *contents* $= r.b$ :: *contents* (Node_2) $= r.a'$ :: *contents* (nothing)

This is a contradiction. This is the same problem dealt with by use of relevant paths in Pierce and Balasubramaniam's model presented in Chapter 2. This means that we need to be certain when reconciling that we are only worrying about those paths whose proper pathprefixes were nonconflicting.

Here is a corrected version with the modifications marked with (*):

```
   fun unisonRecon(o, a, b, a', b': Node) {
     all p: Path | {
       (samesort(p.o::contents, p.a::contents) &&
                    samesort(p.o::contents, p.b::contents))
         => (samesort(p.a::contents, p.a'::contents) &&
                    samesort(p.b::contents, p.b'::contents))
       else (samesort(p.o::contents, p.a::contents) &&
(*)          {all q: Path | PathPrefix(q, p)
(*)            => !conflict(q.o::contents, q.a::contents, q.b::contents)})
                        => (samesort(p.b::contents, p.a'::contents) &&
                             samesort(p.b::contents, p.b'::contents))
       else (samesort(p.o::contents, p.b::contents) &&
(*)          {all q: Path | PathPrefix(q, p)
(*)            => !conflict(q.o::contents, q.a::contents, q.b::contents)})
                        => (samesort(p.a::contents, p.a'::contents) &&
                             samesort(p.a::contents, p.b'::contents))
       else (samesort(p.a::contents, p.a'::contents) &&
                         samesort(p.b::contents, p.b'::contents))
     }
   }
```

This description satisfies the properties of synchronization in an assertion checked in a scope of 6, using the definition of synchronization from the previous section. This assertion claims that if Unison finds some synchronization of two filesystems against an original then two sets of dirty paths *dirtya* and *dirtyb* can be found, satisfying the requirements of exact update detection, such that the result of Unison's synchronization is a synchronization in the sense of the previous section:

```
    assert checkUnisonSafe {
      all o, a, b, a', b': Node | {
        unisonRecon(o, a, b, a', b') =>
          some dirtya, dirtyb: set Path | {
            dirty(o, dirtya, a)
            dirty(o, dirtyb, b)
            synchronization(a, b, a', b', dirtya, dirtyb)
          }
      }
    }
}
```

This gives us some confidence that the Unison spec describes a correct synchronizer. We also checked in a scope of 6 that the description actually synchronizes all filesystems that it should; if a synchronization of two filesystems against an original exists in the scope being considered, given an exact update detector, then the description of Unison finds the synchronization as well:

```
assert unisonMaximal {
  all o, a, b, a', b': Node, dirtya, dirtyb: set Path | {
    (dirty(o, dirtya, a) &&
     dirty(o, dirtyb, b) &&
     synchronization(a, b, a', b', dirtya, dirtyb)) =>
         unisonRecon(o, a, b, a', b')
  }
}
```

What is interesting to note in this model is the difference in style between it and the previous model of synchronization; the specification of Unison update detection is written differently from the update detection in section 3.2.2 for the original model of synchronization that was examined. However, this was in keeping with what was noted in the description of update detection earlier; very different policies of update detection are easily examined by substituting a new description of dirtiness into the model.

Also of interest is the difference between the reconciliation portions of synchronization in either model. The Unison specification model adopts a very declarative approach, describing properties of the results of reconciliation rather than modeling the algorithm that would achieve it as in the previous chapter.

This kind of flexibility in description is valuable for determining not only if an algorithm is correct but also if your understanding of the problem is. As the counterexample above demonstrates, it is very easy to write a model of some simple sounding properties and misinterpret the English language description of those properties or miss a subtlety that was not stated.

The complete text of this model can be found in Appendix B.4.

## 4.2 Briefcase Specification

Briefcase is a widely used piece of synchronization software that ships with most modern versions of Microsoft Windows. While it is difficult to find a description of Briefcase more detailed than "Briefcase automatically updates files", some exist. On the Microsoft website [9], the Briefcase specification is informally described in a case by case manner as follows:

> 1) When you synchronize files, the files that you opened or updated while disconnected from the network are compared to the versions of the files that are saved on the network. As long as the same files you changed haven't been changed by someone else while you were offline, your changes are copied to the network.
>
> 2) If someone else made changes to the same network file that you updated offline, you are given a choice of keeping your version, keeping the one on the network, or keeping both. To save both versions of the file, give your version a different file name, and both files will appear in both locations.
>
> 3) If you delete a network file on your computer while working offline but someone else on the network makes changes to that file, the file is deleted from your computer but not from the network.
>
> 4) If you change a network file while working offline but someone else on the network deletes that file, you can choose to save your version onto the network or delete it from your computer.
>
> 5) If you are disconnected from the network when a new file is added to a shared network folder that you have made available offline, that new file will be added to your computer when you reconnect and synchronize.

This description is more difficult to model since the tasks of update detection and reconciliation are described together and there are ambiguities in the description. Also, the description is incomplete. It is not stated that if you do not change a file and somebody else does that you will ever receive an updated version of the file (a basic example of nonconflicting update).

Also, note that deleting a file is normally considered by users to be a change to that file and yet the description treats file modifications and deletions in completely different ways. As written, Briefcase does not satisfy requirements of synchronization since certain updates are considered more meaningful than others (deletions are ignored for propagation in favour of other modifications). Briefcase attempts synchronization while trying to remain conscious of user's views on a filesystem and, as one result, successful synchronizations in the absence of conflicts do not have to result in identical filesystems.

### 4.2.1 Client-side Synchronizations

Briefcase can be used in another manner that the description above does not discuss. Rather than initiating synchronization from the networked version of a filesystem and simultaneously synchronizing two filesystems against an original as in Unison, Briefcase can be initiated by a client to synchronize only its replica with a networked version of the filesystem that records some changes in replicas from the original version.

Experimentation reveals that this client-side use of Briefcase actually does not satisfy the fourth requirement and treat deletions as low priority changes; instead, it treats deletions as modifications (as in Unison) and follows the first requirement where changes made to a file on one replica after the file has been deleted on another do not get propagated. After these points are considered, a reasonable interpretation for client-side use of Briefcase, assuming no user interaction, would be as follows:

> 1) When you synchronize files, the files that you updated (added, deleted, changed) while disconnected from the network are compared to the versions of the files that are saved on the network. As long as your updates aren't conflicting with another user's changes, your changes are copied to the network.
>
> 2) If someone else made changes to the same network file that you updated offline, you keep your version and leave the network version alone.
>
> 3) If you are disconnected from the network when a new file is added or some file you have not updated is changed, that file will be added to (or replaced on) your computer when you synchronize.

This version seems to describe the same thing the Unison specification describes: propagate non-conflicting updates and stop at conflicts. However, Unison can synchronize arbitrary pairs of filesystems while Briefcase in this mode can only synchronize one replica against the network version of the filesystem. It seems natural to ask if this mode of Briefcase can be used to simulate the behaviour of Unison, with the networked version of the filesystem functioning as go-between.

### 4.2.2 Simulating Unison with Briefcase

If we examine an iterated run of Briefcase where we synchronize the first replica with the network version, the second replica with the network version, and then the updated first replica with the network version again, we can check in our model that the result is the same as having synchronized both replicas at once through Unison.

Since Briefcase's specification seems so similar to Unison's, it makes sense to consider an alternative use of the function modeling Unison's reconciliation specification described in the previous section. We will use the *unisonRecon* function to describe steps of Briefcase and check that there is an equivalence to Unison's behaviour. Given an original network version of a filesystem $n$ and two replicas $a$ and $b$ we check that Briefcase simulates Unison using the following assertion:

```
    assert briefcase {
     all n, a, n1, a1, b, b1, n2, a2, n3: Node | {
(1)       {unisonRecon(n, a, n, a1, n1)
(2)        unisonRecon(n, b, n1, b1, n2)
(3)        unisonRecon(a1, a1, n2, a2, n3)}
(4)            => unisonRecon(n, a, b, a2, b1)
     }
    }
```

Line 1 corresponds to synchronizing $a$ with the network, $n$, and against its original (the original version of $a$ and $b$ is assumed to be the first network version). It results in a new, synchronized version of $a$ called $a1$ and a new network version called $n1$.

Line 2 corresponds to synchronizing $b$ with the network, $n1$, and against its original, $n$. It results in a new, synchronized version of $b$ called $b1$ and a new network version, $n2$.

Line 3 corresponds to synchronizing the synchronization of $a$, $a1$, with the network version, $n2$, and against its last synchronization, $a1$. This results in a new, synchronized version of $a1$ called $a2$ and a new network version called $n3$.

Lines 1, 2, and 3 together imply Line 4. Line 4 describes Unison's behaviour when synchronizing two replicas $a$ and $b$ against an original version called $n$. This results in $a$ being updated to $a2$ and $b$ being updated to $b1$.

The assertion seems to reveal an equivalence to Unison. This is surprising since the descriptions of synchronization used by either software package and the styles of synchronization (simultaneous synchronization versus serialized synchronization) sound very different. It seems that the basic ideas behind policies for synchronization tend to be shared.

### 4.2.3 A Limitation of the Unison/Briefcase Analogy

This method of describing Briefcase breaks down after the three step iteration. After the three step iteration is completed, consider immediately synchronizing the synchronized filesystems, without making any changes to them. Common sense tells us that the expected behaviour would be to have no changes made to the filesystems. However, examine what happens if we continue the line of reasoning from above.

We synchronize the synchronized first replica, $a2$, with the latest network version, $n3$, to get a new replica, $a3$, and new network filesystem, $n4$. Then we synchronize the synchronized second replica, $b1$, with $n4$. Finally, we synchronize the synchronized first replica, $a3$, with the network version of the filesystem again. What will the results be? Consider the following valid assertion:

```
assert mirroring {
   all f, g: Node | unisonRecon(f, f, g, g, g)
}
```

This assertion tells us that when one replica of a filesystem is identical to the original version of the filesystem and another replica is different, then *unisonRecon* propagates the changes from the differing replica to the unchanged replica. This has some interesting consequences in our examination of Briefcase.

When we synchronize $b1$ with $n4$, we are synchronizing against the last synchronized version of $b$, which happens to be $b1$. As a result, by mirroring, we know that $b1$ will be replaced with the filesystem $n4$. If we apply the mirroring property to previous steps, we find that $n4$ is actually the same as $a2$. This means that synchronizing again after no changes are made to the synchronized filesystems results in the synchronized replica of $b$ being replaced with the synchronized replica of $a$ while the synchronized replica of $a$ remains the same. This is a problem since there is potential for data to be lost.

This is not the whole story about Briefcase synchronization. Experimentation reveals that there is extra state stored by Briefcase that prevents one from being able to really understand its behaviour from its specification. For example, if a file is deleted on one filesystem replica and the replica is synchronized with the network version, causing the file to be deleted from the network version, then an updated copy of that file on another replica will not be propagated to the network version unless the file is deleted from that replica, the replica is synchronized, the file is replaced, and the replica is synchronized again. This indicates that there is state stored that is not described in [9].

Also, Briefcase experiments reveal a laziness in the treatment of directories. Directory structure changes that don't affect paths of files are not propagated unless there is already some other change being propagated. This is interesting and potentially frustrating since it is conceivable that directory structure can contain a large amount of information.

It is apparent that while descriptions of properties that should be satisfied by synchronization may sound similar, the number of implementation details, ambiguities in the descriptions, and the urge to make assumptions about users' wants and needs can lead to numerous synchronization programs with very different behaviour.

Alloy models in full for this section can be found in Appendix B.4.

# Chapter 5

# Conclusions

Constructing models of filesystem synchronization properties and algorithms served to outline similarities and differences in different synchronization policies. Reasonable sounding policies can be easily misinterpreted or even be contradictory. Since synchronization is such an important task it is important to understand what happens when synchronization is initiated.

I stated and checked using the Alloy analyzer all listed propositions about synchronization described in [2, 1], including the key properties that there can only be one maximal synchronization of filesystems $A$ and $B$ against an original filesystem $O$ and that the published reconciliation algorithm employed by Unison calculated that unique maximal synchronization.

I discovered that the Unison synchronizer (based on formalized notions of filesystem synchronization) itself has an informal description of its specification in its user's manual that suffers from an inconsistent description of reconciliation, leading to an easily believed, but incorrect, understanding of what Unison accomplishes. I also modeled an operational description of Briefcase's synchronization specification only to discover after some reformulation that it claims the same policy as Unison. However, there are many published examples of "unusual" behaviour by Briefcase that serve to demonstrate that while a spec might be available it is not a reliable description of actual behavior of software.

Models of policies as well as algorithms proved to be very succinct. A reference implementation of Unison is given in [10] and stands at about 450 lines in length while the core Alloy model of the Unison algorithm is only about 60 lines (the model of the specification is even shorter). While Pierce and Vouillon's implementation is useful (as the large Unison user population can attest to), Alloy models are smaller and provide a more lightweight alternative to examination of synchronization policies. Many details of filesystems and the mechanics of their operation were not important in the

Alloy model, allowing for shorter descriptions of synchronization. Also, through interactive use of the Alloy analyzer, redundant restrictions and policies in the originally constructed Alloy model were easily eliminated. All these point encourage treatment of modeling as an important step in system development.

One of the limitations of the models presented is the machinations necessary to overcome Alloy's lack of recursive function support. I originally constructed an algorithmic description of reconciliation that mimicked almost exactly the algorithm as presented in [2]. However, it required manual unfolding of the recursion to the required depth for any given test. A clean declarative specification was less easy to write; the recursion had to be hidden using clever reformulations of reconciliation while being careful not to overconstrain the filesystems being synchronized. This did lead to a better understanding of the whys of the reconciliation algorithm, but presented a hurdle when first attempting to model the algorithm.

Alloy itself presented other challenges to modeling. Use of functions in Alloy must be done carefully to not introduce unnecessarily large formulas for the SAT solvers to handle. In some cases this was not entirely trivial and reduced readability of the models. Also, Alloy's use of finite scope, while a boon in offering automated analysis, introduces issues related to defining models that can be confusing to users. It can be subtle to express properties, such as the natural one that appending one path to another creates a new path since at some point there are not enough path atoms in a finite model to combine other paths to. Since the property, if written as a fact, needs an infinite number of paths to be true, any model constructed by Alloy will have no paths at all.

One of the greatest challenges was not an artifact of Alloy or of filesystem synchronization but instead was just determining when a model had captured the essence of a problem or provided some useful information. Alloy helps with this since its declarative statements allow modeling at almost any level of abstraction or detail that you might want, as demonstrated in several models here (the first model describes a particular algorithm in detail while the Unison model describes a specification for synchronization). Filesystem synchronization proved both subtle and varied but the modeling task itself became the most interesting endeavor.

# Bibliography

[1] S. Balasubramaniam and B. C. Pierce. File synchronization. Technical Report 507, Computer Science Department, Indiana University, Apr. 1998.

[2] 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.

[3] The Coq Proof Assistant. http:///pauillac.inria.fr/coq/, May 2002.

[4] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering, Special Issue on Formal Methods in Software Practice*, 23(5), May 1997.

[5] D. Jackson. Micromodels of software: Lightweight modelling and analysis with alloy. Reference Manual; available through http://sdg.lcs.mit.edu/alloy/, 2001, 2002.

[6] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)*, Sept. 2001.

[7] D. Jackson and K. Sullivan. Com revisited: Tool assisted modelling and analysis of software structures. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, San Diego, Nov. 2000.

[8] S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, Sept. 2000.

[9] Microsoft Corporation. Handling File Conflicts. http://www.microsoft.com/windows2000/en/professional/help/csc_handle_file_conflicts.htm, February 2000.

[10] B. C. Pierce and J. Vouillon. Unison: A file synchronizer and its specification. Technical report; available through http://www.cis.upenn.edu/ bcpierce, 2001.

[11] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. *Foundations of Software Engineering*, 2001.

[12] Unison File Synchronizer User Manual and Reference. http://www.cis.upenn.edu/ bcpierce/unison/manual.html#updates.

# Appendix A

# Tutorial Introduction to Modeling Synchronization

This tutorial describes in detail the process of modeling Pierce and Balasubramaniam's two part (update detection and reconciliation) formalization [2] of filesystem synchronization in Alloy, a lightweight formal modeling language. This tutorial does not go into detail about the process of synchronization, but it includes enough information about the task to follow development of the model. This document assumes knowledge of the basic syntax and semantics of the Alloy language [6] and is intended to serve as an example of development and troubleshooting of a complex model.

## A.1 Filesystems

The first step to modeling filesystem synchronization is to model filesystems. Since synchronization deals with names, paths, and filesystem nodes it makes sense to introduce three different kinds of atoms: Names, Paths, and Nodes. Names will identify files and directories and paths will be sequences of names. Nodes are filesystem nodes (files and directories).

```
sig Name {}
sig Path {
  names: Seq[Name]
}
sig Node {}
```

The *seq* library included with the Alloy distribution is used to define Paths, as can be seen above.

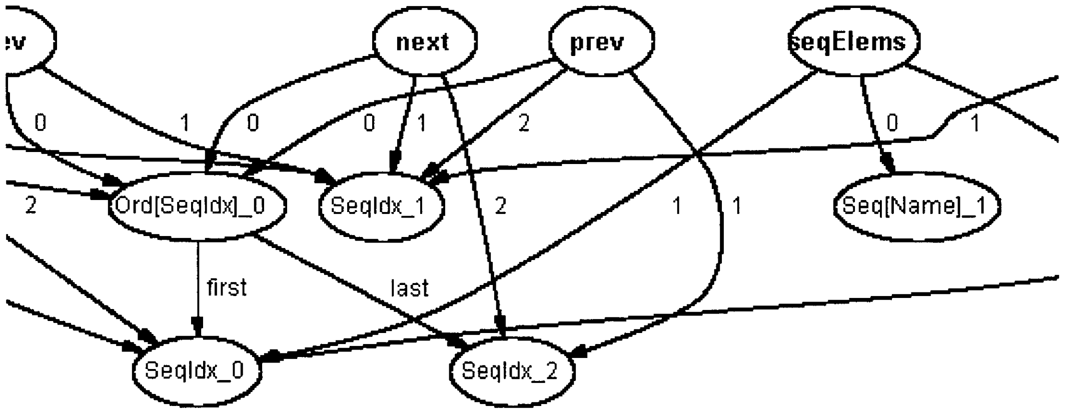
Nodes will be either files or filesystems:

```
disj sig File extends Node {}
disj sig Filesystem extends Node {}
fact FileDirPartition {
  File + Filesystem = Node
}
```

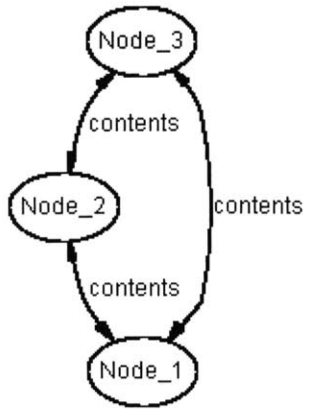
Synchronization is described most easily using paths. To facilitate this style of description, we'll make filesystems partial mappings from paths to other nodes, rather than partial mappings from names to other nodes. We add a *contents* relation to the Filesystem signature.

```
disj sig Filesystem extends Node {
  contents: Path -> Node
}
```

Now let's take a look at just part of an example Alloy generates. This example was generated and visualized automatically by the Alloy analyzer. The visualization was not customized in any way:

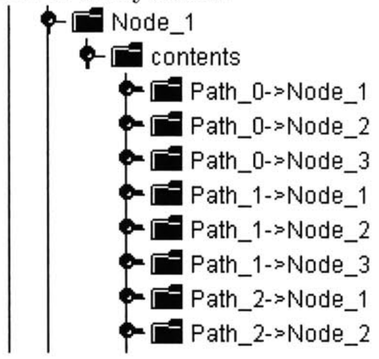


It's hard to figure out exactly what's going on here, so we change the customization to make it easier to read examples. We only display Nodes and the *contents* relation:



Now it's possible to easily see what other nodes each filesystem maps to. Remember, the *contents* relation maps paths to nodes, rather than names to nodes, meaning that a filesystem node will have an arrow leading to any node that is reachable from it.

This almost looks believable as a filesystem, but let's take a closer look at what the paths actually are. This is a screenshot of the Alloy analyzer's "Solution" portion of its display, expanded to view one node's contents in an automatically generated model of filesystems:



```
Node_1
  contents
    Path_0->Node_1
    Path_0->Node_2
    Path_0->Node_3
    Path_1->Node_1
    Path_1->Node_2
    Path_1->Node_3
    Path_2->Node_1
    Path_2->Node_2
```

This is problematic. Note that Node 1's contents map the same path to multiple nodes (such as Path 0 mapping to Nodes 1, 2, and 3), as well as mapping each path to at least one node.

This leads to a change in the definition of filesystems:

```
fun EmptyPath (p: Path) {
  p.names..SeqIsEmpty()
}
disj sig Filesystem extends Node {
  contents: Path ? -> ? Node
  } {
    some p: Path | EmptyPath(p) && this = p.contents
  }
```

Note the multiplicity constraints expressing that contents are a partial mapping from some paths to at most one node. Also, note the additional constraint ensuring that following the empty path from the node leads to itself. The statement also ensures that there is an empty path. Empty paths are defined as paths where the sequence of names it represents is empty.

Next, when generating models we find that it is possible for two different paths to be comprised of the same sequence of names. This is common in Alloy models and for simplicity's sake we want to canonicalize the atoms that are being considered:

```
fact CanonicalPath {all a, b: Path| a.names = b.names => a = b}
fact CanonicalSequence {all a, b: Seq[Name] | a..SeqEquals(b)
                                                    => a = b}
```

A similar issue exists with nodes. To canonicalize nodes, we first have to include a notion of equivalence for nodes expressing that two nodes are equivalent if they are the same node or if they are filesystems mapping the same paths such that if a path is mapped to a file in one then it is mapped to the same file in the other. Note the use of *result* in the *Paths* function. *Paths* "returns" a set of *Paths*, the value of which is the set of paths mapped by the filesystem's *contents* relation. For memory reasons it is preferable to write the body of functions such as *Paths* that are used as expressions as "*result* =" whenever possible:

```
fun Paths(f: Filesystem): set Path {
  result = Node.~(f.contents)
}
fun EquivNode (f, g: Node) {
  (f = g) ||
  (f in Filesystem && g in Filesystem && Paths(f) = Paths(g) &&
  all p: Path | all e: File | {
    p->e in f.contents <=> p->e in g.contents
  })
}
fact CanonicalNode {all a, b: Node | EquivNode(a, b) => a = b}
```

We need to start considering restrictions to the atoms to make sure that filesystems behave as we expect. For example, this seems like a reasonable description: for any paths $p$ and $q$, looking up a composite path $p.q$ is the same as first looking up the first portion of the path, $p$, and then looking up the remainder of the path, $q$, from the resulting sub-filesystem. To express this constraint we need to describe the append operation. This requires using functions from the standard sequences library because paths were defined as sequences. First, we write a description for path length:

```
fun PathLength (p: Path): set SeqIdx {
  result = SeqInds(p.names)
}
```

This description just returns the set of indices used to define the path. Next we define what it means to be a pathprefix by stating that $a$ is a prefix of $b$ if $b$ is a sequence starting with $a$:

```
fun PathPrefix (a, b: Path) {
  b.names..SeqStartsWith(a.names)
}
```

With these two functions we can define the append operation. The *append* function returns a path of length equal to that of the sum of the input paths' lengths. It also specifies that input path $a$ is a prefix of the result and that for all indices past those of $a$, the appropriate values from $b$ are present. The # operator returns the cardinality of the set it precedes.

```
fun append (a, b: Path): Path {
  result = {r: Path | {
  #PathLength(r) = #PathLength(a) + #PathLength(b)
  PathPrefix(a, r)
  all i: r.names..SeqInds() | {
    #OrdPrevs(i) >= #PathLength(a) =>
    some k: SeqIdx | {
      #OrdPrevs(k) + #PathLength(a)= #OrdPrevs(i)
      b.names..SeqAt(k) = r.names..SeqAt(i)
    }
  }
  }}
}
```

Now we can finally express the composite path property described above:

```
    fact TreeStructure {
      all f: Filesystem | all p, q: Path | {
        let g = p.f::contents | {
          q.g::contents = append(p, q).f::contents
        }
      }
    }
```

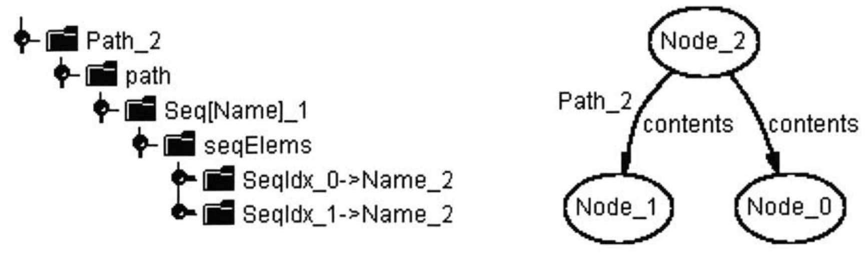
However, there is a problem. After generating several examples, it is conspicuous that files do not seem to be pointed to by any filesystems. To see if this is always the case consider the following function:

```
    fun simple () {some File & Path.Filesystem::contents}
```

When this function is run in the Alloy analyzer for 3 atoms, there are no solutions. The constraint on filesystems disallows existence of files in filesystem contents. Why? Consider a path $p$. Say that in a particular filesystem this path leads to a file. Following the empty path $\varepsilon$ from a file yields nothing since files don't have path contents. Since path $p$ can be expressed as $p.\varepsilon$, the filesystem in question would map path $p$ both to the file (since $p$ leads to a file) and to nothing (since following $p$ in the filesystem is the same as following $p$ to the file it maps to and then following $\varepsilon$ to nothing). As a result, this has to be treated as a special case to be considered in the description of the tree structure constraints (*):

```
    fact TreeStructure {
      all f: Filesystem | all p, q: Path | {
        let g = p.f::contents | {
(*)       (g in File && EmptyPath(q)) ||
             (q.g::contents = append(p, q).f::contents)
        }
      }
    }
```

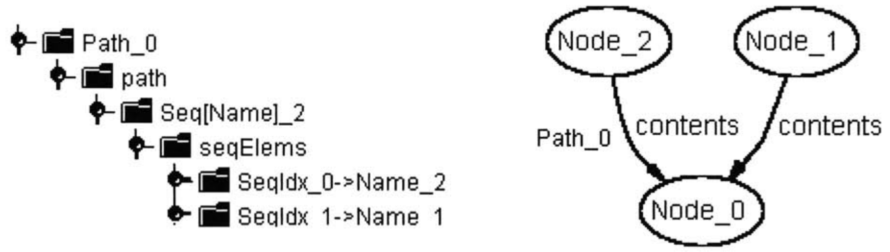
Now let's take another look at an automatically generated filesystem:



This example demonstrates that there is still a problem with filesystem specification. Node 2 leads to Node 1 in one step (the path from Node 2 to Node 1 does not go through a third node) but the path between them is of length 2 (as can be seen in the expansion of the path to the left of the filesystem diagram). The next constraint makes it the case that the immediate pathprefix of a path is also a path. This fact, since true of all paths, will give us that all pathprefixes of a path are paths. $PPathPrefix$ simply defines a proper pathprefix:

```
fun PPathPrefix (a, b: Path) {
  PathPrefix(a, b) && #PathLength(a) != #PathLength(b)
}
fact NoMissingPathBegin {
  all p: Path | {
    !EmptyPath(p) => {
    some q : Path | (PPathPrefix(q, p) &&
                    (#PathLength(p) = #PathLength(q) + 1))
    }
  }
}
```

Here is another example the Alloy analyzer generates:



Notice again that there is a path of length two. This has a similar problem to the one we just fixed, only in the other direction (postfixes). What we actually need is a restriction that all possible subpaths of a path are actually paths as well. We first define *tail* to express almost the same property as that of a tail of a list, using sequence library functions:

```
fun tail (p: Path, q: Path) {
  p.names..SeqRest() = q.names
}
```

Then we constrain the model so that tails of paths exist for nonempty paths:

```
fact NoMissingPathEnd {
  all p: Path | {
    !EmptyPath(p) => {
           some q: Path | tail(p, q)
    }
  }
}
```

To be on the safe side, we actually check that if a path is in a filesystem's contents then any prefix of it also is:

```
assert NoMagicPath {
  all f: Filesystem | all p, q: Path | all n: Node | some m: Node |{
    (PPathPrefix(p, q) && q->n in f.contents) =>
                             (m!=n && p->m in f.contents)
  }
}
```

This produces no counterexamples and generated examples from the model look like expected filesystems. At this point, when trying to find an assertion that breaks the filesystem model, we actually discover an interesting one that happens to be true:

```
assert NoCycles {
  all f: Filesystem | no p: Path | {
    !EmptyPath(p) && p->f in f.contents
  }
}
```

This is true because Alloy works in finite scopes. If it were the case that a cycle could exist, then we would need an infinite number of paths to preserve the TreeStructure fact.

We now move on to the next portion of filesystem synchronization modeling: update detection.

## A.2 Update Detection

Update detection computes a predicate *dirty* on paths s.t. if a path is not dirty in a replica $A$, then it maps to the same thing in the replica as it does in the original filesystem $O$ ($\forall p : Path \mid \neg dirty_A(p) => O(p) = A(p)$). For convenience's sake, dirtiness is up-closed; if path $p$ is a prefix of path $q$ and $q$ is dirty, then $p$ is as well

$(\forall p, q : \mathit{Path} \mid (p \leq q \;\&\&\; \mathit{dirty}_A(q)) \Rightarrow \mathit{dirty}_A(p))$. This makes defining reconciliation easier:

```
fun dirty (f, g: Filesystem): set Path {
  all p: Path | {
    p !in result => p.f::contents = p.g::contents
    else {all q: Path | PathPrefix(q, p) => q in result}
  }
}
```

One fact we want to be sure of is that if a path is not dirty in either of two replicas, then it must be the case that it maps to the same thing in both replicas:

```
assert NonDirtyAreSame {
  all a, b, o: Filesystem | all p: Path |
                            all dirtya, dirtyb: set Path | {
      (dirty(o, dirtya, a) && dirty(o, dirtyb, b) &&
                                   p !in dirtya && p !in dirtyb)
          => p.a::contents = p.b::contents
  }
}
```

There are no counterexamples so we move on to describing reconciliation.

## A.3 Synchronization

Given an original filesystem and two updates $A$ and $B$, synchronization can be described as propagating non-conflicting updates and stopping at conflicts to produce two new filesystems $A'$ and $B'$, updated versions of $A$ and $B$. Updates are changes from the original filesystem and conflicting updates are paths where both replicas differ from the original filesystem and from each other. Informally, the description is a list of conditions:

• If a path $p$ is not dirty in $A$ then any updates in the corresponding subtree in $B$ should be propagated in both $A'$ and $B'$ and similarly for $B$.

• If a path $p$ refers to a directory in both $A$ and $B$ then it should refer to a directory in both $A'$ and $B'$.

• If a path is dirty in both $A$ and $B$ and is not a directory, then there might be a conflict. We stop propagation and keep the value of the paths in $A'$ and $B'$ the same as in $A$ and $B$ respectively.

To express these properties it is useful to define a function indicating whether a particular path leads to a directory in the designated filesystem by checking that the path maps to something in the set of Filesystem atoms:

```
    fun isDir (f: Filesystem, p: Path) {
      some (Filesystem & p.f::contents)
    }
```

Just for convenience, we also add a function indicating whether a path is a directory in two filesystems:

```
    fun isDirAB (f, g: Filesystem, p:Path) {
      isDir(f, p) && isDir(g, p)
    }
```

Now we can translate almost without change the informal description above of properties of synchronization into Alloy:

```
  fun afterPSynchronization (a, b, a', b': Node, dirtya, dirtyb: set Path,
                                                                 p: Path) {
      all pq: Path | {
        PathPrefix(p, pq)
          => {
            pq !in dirtya => (pq.a'::contents = pq.b'::contents &&
                              pq.a'::contents = pq.b::contents)
            pq !in dirtyb => (pq.a'::contents = pq.b'::contents &&
                              pq.a'::contents = pq.a::contents)
            isDirAB(a, b, pq) => isDirAB(a', b', pq)
            (pq in dirtya && pq in dirtyb && !isDirAB(a, b, pq)) =>
                            (pq.a'::contents = pq.a::contents &&
                             pq.b'::contents = pq.b::contents)
           }
      }
    }
```

One property that Pierce and Balasubramaniam are concerned with at this point is that there is only one possible synchronization ($a1$ and $b1$) of two filesystem updates ($a$ and $b$) from an original ($o$), given an update detection policy. Notice the use of the *dirty* function here. The sets *dirtya* and *dirtyb* are passed into *dirty* as the second argument, changing the use of *dirty* from one of expression to one of formula. The dirty function is effectively checking that the set *dirtya* (*dirtyb*) actually is a valid set of dirty paths of *a* (*b*) from *o*:

```
assert Uniqueness {
  all a, b, o: Filesystem | all dirtya, dirtyb: set Path | {
    all p: Path | all a1, b1, a2, b2: Filesystem | {
      (dirty(o, dirtya, a) && dirty(o, dirtyb, b) &&
       afterPSynchronization(a, b, a1, b1, dirtya, dirtyb, p) &&
       afterPSynchronization(a, b, a2, b2, dirtya, dirtyb, p)) =>
                           (p.a1::contents = p.a2::contents &&
                            p.b1::contents = p.b2::contents)
    }
  }
}
```

There are no counterexamples, so we can move on to reconciliation.

## A.4 Reconciliation

What is reconciliation? It is the algorithm employed to arrive at the synchronization of two updated versions of a filesystem. Here is an algorithm from [2] that takes a pair of filesystems and returns a new pair where the subtrees rooted at $p$ have been synchronized:

```
$recon(A, B, p) =$
1) if $\neg dirty_A(p) \bigwedge \neg dirty_B(p)$ then (A, B)
2) else if $isdir_{A,B}(p)$
     then let $\{p_1, p_2, \ldots, p_n\} = children_{A,B}(p)$
        in let $(A_0, B_0) = (A, B)$
           let $(A_{i+1}, B_{i+1}) = recon(A_i, B_i, p_{i+1})$ for $0 \leq i < n$
            in $(A_n, B_n)$
3) else if $\neg dirty_A(p)$ then $(filesystemOverwrite(A, B, p), B)$
4) else if $\neg dirty_B(p)$ then $(A, filesystemOverwrite(B, A, p))$
5) else $(A, B)$
```

$FilesystemOverwrite(A, B, p)$ overwrites filesystem $A$ at path $p$ with $B$ at $p$.

So how will this algorithm be expressed in Alloy? We'll first define some of the functions used in the algorithm. *Children* returns the set of paths that are immediate children of a directory:

```
fun children (f: Filesystem, p: Path): set Path {
  result = {q: Path | {
    q in Node.~f::contents
    PathPrefix(p, q)
    #PathLength(p) + 1 = #PathLength(q)
  }}
}
```

*ChildrenAB* is the set of children of a path $p$ in either filesystem indicated:

```
fun childrenAB (f, g: Filesystem, p: Path): set Path {
  result = (children(f, p) + children(g, p))
}
```

Incomparable paths are ones where neither path is a prefix of the other:

```
fun incomparable (p, q: Path) {
  !PathPrefix(p, q) && !PathPrefix(q, p)
}
```

Overwriting a filesystem $A$ by filesystem $B$ at a path $p$ is described in [2] as
$\lambda q.$if $p \leq q$ then $B(q)$ else $A(q)$
This is straightforward to describe, with the "else" statement in the lambda expression being expressed by the two "else" statements in the code:

```
fun filesystemOverwrite (t, s: Node, p: Path): Node {
  all q: Path | {
    PathPrefix(p, q) => q.result::contents = q.s::contents
    else incomparable(p, q) => q.result::contents = q.t::contents
    else children(result, q) + p = children(t, q) + p
  }
}
```

### A.4.1 Recursive Function Version of *Recon*

The first version of *recon* written stays close to the algorithm just presented.

For use in the definition of *recon*, *firstNode* and *secondNode* break up a pair of nodes into the first or second node respectively (tuple deconstructors):

```
fun firstNode(c: Node -> Node) : Node {
  all a, b: Node | {
      c = a->b => result = a
    }
  }
  fun secondNode(c: Node -> Node) : Node {
    all a, b: Node | {
      c = a->b => result = b
    }
  }
```

Now the reconciliation algorithm can be expressed, closely following the algorithm given above.

```
  fun recon(a, b: Node, p: Path, dirtya, dirtyb: set Path)
                                           : Node->Node {
    one result
    (p !in dirtya && p !in dirtyb)
       => result = a->b
    else isDirAB(a, b, p)
       => reconhelper(a, result, b, dirtya, dirtyb,
                                  childrenAB(a, b, p))
    else p !in dirtya
       => {some abp: Node | {filesystemOverwrite(a, abp, b, p)
                                      && result = abp->b}}
    else p !in dirtyb
       => {some bap: Node | {filesystemOverwrite(b, bap, a, p)
                                      && result = a->bap}}
    else result = a->b
  }
```

*Reconhelper* is the same as *recon*, except it is written to deal with synchronizing a number of sibling paths, rather than just one path:

```
  fun reconhelper(a, b: Node, dirtya, dirtyb: set Path,
                        childs: set Path): Node->Node {
    one result
    no childs => result = a->b
    else some q in childs | some c: Node->Node | {
      one q
      let c = recon(a, b, q, dirtya, dirtyb) | {
        reconhelper(firstNode(c), result, secondNode(c),
                             dirtya, dirtyb, (childs - q))
      }
    }
  }
```

### A.4.2 Manually Unrolled Recursive Function Version

Alloy does not currently support recursion. As a result, the model of recursion from above has to be manually unfolded by copying the recursive functions multiple times and chaining them together. In the interests of developing a more declarative style version of reconciliation, here is another attempt at a description, where $a'$ and $b'$ are introduced as arguments representing results of synchronizations:

```
fun recon(a, b, a', b': Node, p: Path, dirtya, dirtyb: set Path) {
  (p !in dirtya && p !in dirtyb)
    => (p.a'::contents = p.a::contents && p.b'::contents =
                                                 p.b::contents)
  else isDirAB(a, b, p)
    => {no childrenAB(a, b, p) =>
                    (p.a'::contents = p.a::contents &&
                     p.b'::contents = p.b::contents)
  else all q in childrenAB(a, b, p) | {
    recon2(a, b, a', b', q, dirtya, dirtyb)
  }}
  else p !in dirtya
    => (p.b'::contents = p.b::contents
                              && filesystemOverwrite(a, a', b, p))
  else p !in dirtyb
    => (p.a'::contents = p.a::contents
                              && filesystemOverwrite(b, b', a, p))
  else (p.a'::contents = p.a::contents &&
                                p.b'::contents = p.b::contents)
}
```

*Recon*2 is simply a copy of *recon* which in turn calls a *recon*3, etc. However, this model heavily overconstrains synchronizations. Why? *FilesystemOverwrite* constrains the result of reconciliations to be exactly an input filesystem overwritten at one path by another filesystem. However, in general, *filesystemOverwrite* is called several times in the course of one reconciliation, on different paths. These cases are not considered if this description is used. To rectify this, we first loosen the definition of *filesystemOverwrite* to only concern itself with the point of the overwrite and beyond:

```
    fun filesystemOverwrite (t, s: Node, p: Path): Node {
      all q: Path | {
        PathPrefix(p, q) => q.result::contents = q.s::contents
      }
    }
```

Then we use the same recon function from above, only we precede it with constraints on paths outside the point of synchronization to prevent underconstraint(*):

```
    fun recon(a, b, a', b': Node, p: Path, dirtya, dirtyb: set Path) {
(*)   all q: Path | {
(*)     incomparable(p, q) => (q.a'::contents = q.a::contents &&
(*)                            q.b'::contents = q.b::contents)
(*)     (q !in Paths(a) && q !in Paths(b)) =>
(*)                 (q !in Paths(a') && q !in Paths(b'))
(*)     (q in Paths(a) && q in Paths(b)) =>
(*)                 (q in Paths(a') && q in Paths(b'))
(*)     isDirAB(a, b, q) => isDirAB(a', b', q)
(*)   }
      (p !in Paths(a) && p !in Paths(b))
         => (a' = a && b' = b)
      else (p !in dirtya && p !in dirtyb)
         => (a' = a && b' = b)
      else isDirAB(a, b, p)
         => {no childrenAB(a, b, p) => (a' = a && b' = b)
             else all q in childrenAB(a, b, p) | {
               reconhelper(a, b, a', b', q, dirtya, dirtyb)
            }}
      else p !in dirtya
         => (b' = b && filesystemOverwrite(a, a', b, p))
      else p !in dirtyb
         => (a' = a && filesystemOverwrite(b, b', a, p))
      else (a' = a && b' = b)
    }
```

The *reconhelper* function, called by *recon*, is just *recon* without these constraints on paths outside the point of synchronization:

```
fun reconhelper(a, b, a', b': Node, p: Path, dirtya, dirtyb: set Path) {
    (p !in dirtya && p !in dirtyb)
       => (p.a'::contents = p.a::contents && p.b'::contents =
                                             p.b::contents)
    else isDirAB(a, b, p)
       => {no childrenAB(a, b, p) => (p.a'::contents = p.a::contents &&
                                     p.b'::contents = p.b::contents)
    else all q in childrenAB(a, b, p) | {
      reconhelper2(a, b, a', b', q, dirtya, dirtyb)
    }}
    else p !in dirtya
       => (p.b'::contents = p.b::contents
                                   && filesystemOverwrite(a, a', b, p))
    else p !in dirtyb
       => (p.a'::contents = p.a::contents
                                   && filesystemOverwrite(b, b', a, p))
    else (p.a'::contents = p.a::contents &&
                                p.b'::contents = p.b::contents)
  }
```

## A.4.3 A Problem with Synchronization

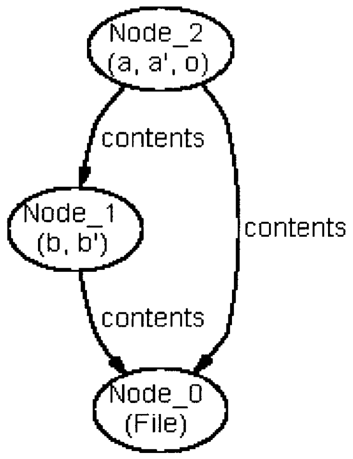
Now that *recon* is written, it is important to start checking its correctness to prevent too much time being spent refining an incorrect model. The obvious thing to check is that it actually produces synchronizations. A synchronization can be expressed as an *afterPSynchronization* where paths outside the point of synchronization remain unchanged in the filesystems being synchronized:

```
fun syncp (p: Path, a', b', a, b: Node, dirtya, dirtyb: set Path) {
   afterPSynchronization(a, b, a', b', dirtya, dirtyb, p)
   all q: Path | {
     incomparable(p, q) =>
        (q.a'::contents = q.a::contents &&
         q.b'::contents = q.b::contents)
     (PathPrefix(q, p) && isDirAB(a, b, q)) => isDirAB(a', b', q)
   }
}
```

Soundness is a straightforward property indicating that if the reconciliation function claims that $a'$ and $b'$ are valid synchronizations of $a$ and $b$ then they actually are:

```
assert Soundness {
  all a, b, o: Filesystem | all dirtya, dirtyb: set Path | {
    all z: Path | all a', b': Filesystem | {
      (dirty(o, dirtya, a) && dirty(o, dirtyb, b) &&
       recon(a, b, a', b', z, dirtya, dirtyb)) =>
             (syncp(z, a', b', a, b, dirtya, dirtyb))
    }
  }
}
```

However, here is a counterexample where filesystems $a$ and $b$ are synchronized from an original $o$ after the empty path. The dirty paths of $a$ include the empty path and the path of length 1 mapping to Node 1. Using the customization settings in visualization we label the nodes with o, a, b, etc. to make it easy to see what is happening:



This counterexample actually demonstrates an error in the description of synchronization.

There are cases where predictions about the results of synchronization are contradictory, as in this example. According to the first property of synchronization in section A.3, the length two path mapping Node 2 to Node 0 (which is not dirty in $a$) should map to the same thing in filesystems $b$, $a'$, and $b'$, namely nothing. However, the length one path mapping Node 2 to Node 1 and Node 1 to Node 0 is dirty in both $a$ and $b$. The third rule of synchronization tells us it should map to the same thing in $a$ and $a'$ (Node 1) and the same in $b$ and $b'$ (Node 0). However, if $a'$ maps to Node 1 and Node 1 contains a path to Node 0, then $a'$ contains a length two path to Node 0, contradicting that it maps the length two path to nothing.

There is an ambiguity in the definition of synchronization that can be resolved by stopping at the first sight of conflict. We introduce a new notion of relevant paths for two filesystems (paths where all ancestors correspond to directories in both filesystems):

```
    fun relevant (a, b: Filesystem, p: Path) {
      all q: Path | {
        PPathPrefix(q, p) => isDirAB(a, b, q)
      }
    }
```

Next, we add the notion of relevant paths to our definition of synchronization (*):

```
 fun afterPSynchronization (a, b, a', b': Node, dirtya, dirtyb: set Path,
                                                                p: Path) {
(*)   relevant(a, b, p)
      all pq: Path | {
(*)     (PathPrefix(p, pq) && relevant(a, b, pq))
           => {
          pq !in dirtya => (pq.a'::contents = pq.b'::contents &&
                             pq.a'::contents = pq.b::contents)
          pq !in dirtyb => (pq.a'::contents = pq.b'::contents &&
                             pq.a'::contents = pq.a::contents)
          isDirAB(a, b, pq) => isDirAB(a', b', pq)
          (pq in dirtya && pq in dirtyb && !isDirAB(a, b, pq)) =>
                            (pq.a'::contents = pq.a::contents &&
                             pq.b'::contents = pq.b::contents)
          }
      }
    }
```

We also add the notion to the property being checked (*):

```
    assert Soundness {
      all a, b, o: Filesystem | all dirtya, dirtyb: set Path | {
        all z: Path | all a', b': Filesystem | {
         (dirty(o, dirtya, a) && dirty(o, dirtyb, b) &&
(*)       relevant(a, b, z) && recon(a, b, a', b', z, dirtya, dirtyb)) =>
                       (syncp(z, a', b', a, b, dirtya, dirtyb))
        }
      }
    }
```

This produces no counterexamples, encouraging confidence in the model of synchronization so far, leading us to ask the question, "How can this model be better?".

### A.4.4 Relational Model of *Recon*

The manual unfolding required to accomplish recursion is not very elegant and a solution that works around this problem is desirable. One solution can be found in introducing a new kind of atom. We call it a *Synchronizer* and it contains two relations that accomplish update detection (mapping an original node $o$ and a replica node $a$ to a set of *DirtyPaths*) and reconciliation (mapping two replica nodes, a path, and the sets of dirty paths to a pair of synchronized nodes):

```
sig DirtyPaths {
  paths: set Path
}
static sig Synchronizer {
  recon: Node -> Node -> Path -> DirtyPaths -> DirtyPaths ->
                                                  !(Node->Node),
  dirty : Node -> Node -> DirtyPaths
}
```

Note the use of *static* above. It marks a synchronizer as a special atom, one of a kind. *DirtyPaths* are introduced as a way to wrap sets of paths representing dirty sets into one datatype so that it is easily used in the *recon* relation. As a result, we also have a *dirty* relation which calculates the set of dirty paths from two nodes. This, of course, means that the definition of dirtiness changes from the original formulation to look like this:

```
fact DirtyBehaviour {
  all f, g: Filesystem | all p: Path | {
    p !in Synchronizer.dirty[f][g].paths =>
                           p.f::contents = p.g::contents
    }
  }
  fact DirtyUpClosed {
    all f, g: Filesystem | all p, q: Path | {
      (PathPrefix(p, q) && q in Synchronizer.dirty[f][g].paths) =>
                            p in Synchronizer.dirty[f][g].paths
    }
  }
```

As you can tell, this is very similar to the previous definition. After dirtiness is described, it remains to describe reconciliation. For the most part, the previous description of reconciliation doesn't have to change much. The difference lies in the treatment of the area requiring recursion. Here, some thought is required to figure out exactly how to model the recursive behavior of reconciliation relationally. The key is to note that there is a relationship between reconciling a particular directory in two filesystems and reconciling any children of those directories.

When a directory is reconciled, the result of that reconciliation at any child path will be the same as the result of reconciling the child path in those directories. With this information in hand, it is straightforward to translate the previous version of reconciliation into a new relational one. No changes are made to the model except in the description of recursion (*). The filesystems $c$ and $d$ in the recursive portion refer to synchronizations of children directories:

```
  fact reconFacts {
    all a, b: Node | all p: Path | all dirtya, dirtyb: DirtyPaths |
                                                some abp, bap: Node | {
      let a' = Node.~(Synchronizer::recon[a][b][p][dirtya][dirtyb]),
          b' = Synchronizer.recon[a][b][p][dirtya][dirtyb][a']  | {
        all q: Path | {
           incomparable(p, q) => (q.a'::contents = q.a::contents &&
                                  q.b'::contents = q.b::contents)
           (q !in Paths(a) && q !in Paths(b)) =>
                               (q !in Paths(a') && q !in Paths(b'))
           (q in Paths(a) && q in Paths(b)) =>
                               (q in Paths(a') && q in Paths(b'))
        }
       (p !in Paths(a) && p !in Paths(b))
         => (a' = a && b' = b)
       (p !in dirtya.paths && p !in dirtyb.paths)
```

```
           => (p.a'::contents = p.a::contents && p.b'::contents =
                                                    p.b::contents))
         else isDirAB(a, b, p)
           => {
           no childrenAB(a, b, p) => (p.a'::contents = p.a::contents &&
                                      p.b'::contents = p.b::contents)
           else all q in childrenAB(a, b, p) | {
(*)          let c = Node.~(Synchronizer.recon[a][b][q][dirtya][dirtyb]),
(*)              d = Synchronizer.recon[a][b][q][dirtya][dirtyb][c]  | {
(*)               (q.a'::contents = q.c::contents &&
(*)                q.b'::contents = q.d::contents)
(*)          }
           }
         }
         else p !in dirtya.paths
           => (p.b'::contents = p.b::contents &&
                       filesystemOverwrite(a, a', b, p))
         else p !in dirtyb.paths
           => (p.a'::contents = p.a::contents &&
                       filesystemOverwrite(b, b', a, p))
         else (p.a'::contents = p.a::contents &&
                     p.b'::contents = p.b::contents)
        }
      }
    }
```

This version follows the structure of the previous almost exactly until the recursive portion. However, as you might imagine, this relation is very large.

### A.4.5 Shrinking the Relational Model

It is an interesting question whether or not the above relation can be smaller in size. One way it can be trimmed is to remove the DirtyPaths and instead pass in the original filesystem $O$:

```
static sig Synchronizer {
  recon: Node -> Node -> Node -> Path -> !(Node->Node),
  dirty : Node -> Node -> Path
}
```

The *recon* relation is one place smaller and dirtiness can now be described without wrapping the result into a new atom since the *dirty* relation stores the information about which paths have been determined dirty. The dirtiness constraints just have to be changed to deal with the *DirtyPaths* fields (*):

```
    fact DirtyBehaviour {
      all f, g: Filesystem | all p: Path | {
(*)         p !in Synchronizer.dirty[f][g] =>  p.f::contents = p.g::contents
      }
    }
    fact DirtyUpClosed {
      all f, g: Filesystem | all p, q: Path | {
(*)     (PathPrefix(p, q) && q in Synchronizer.dirty[f][g]) =>
(*)                             p in Synchronizer.dirty[f][g]
      }
    }
```

Reconciliation also looks almost exactly the same, using *o* as an argument that can be used to look up dirty paths without passing them in:

```
  fact reconFacts {
    all o, a, b: Node | all p: Path | some abp, bap: Node |{
      let dirtya = Synchronizer.dirty[o][a], dirtyb =
                    Synchronizer.dirty[o][b] | {
        (p !in dirtya && p !in dirtyb)
              => Synchronizer.recon[o][a][b][p] = a->b
        else isDirAB(a, b, p)
              => {
         no childrenAB(a, b, p) => Synchronizer.recon[o][a][b][p] = a->b
         all q: childrenAB(a, b, p) | {
           let a' = Node.~(Synchronizer::recon[o][a][b][p]),
               b' = Synchronizer.recon[o][a][b][p][a'],
               c = Node.~(Synchronizer.recon[o][a][b][q]),
               d = Synchronizer.recon[o][a][b][q][c] | {
                (children(a', p) + children(b', p)) in childrenAB(a, b, p)
                 all s: Path | {PathPrefix(s, q) =>
                        ((s in Paths(a') <=> s in Paths(c)) &&
                         (s in Paths(b') <=> s in Paths(d))) }
                (q.a'::contents = q.c::contents && q.b'::contents =
                                                      q.d::contents)
                  }
              }
            }
          else p !in dirtya
               => filesystemOverwrite(a, abp, b, p) &&
                     Synchronizer.recon[o][a][b][p] = abp->b
          else p !in dirtyb
               => filesystemOverwrite(b, bap, a, p) &&
                     Synchronizer.recon[o][a][b][p] = a->bap
          else Synchronizer.recon[o][a][b][p] = a->b
        }
      }
    }
```

This relation is smaller and hence checkable in a larger scope, leading us into testing.

### A.4.6 Testing Reconciliation

The soundness property looks almost exactly the same as before, except referring to relations to accomplish update detection and reconciliation:

```
assert Soundness {
  all a, b, o: Filesystem | {
   let dirtya = Synchronizer.dirty[o][a],
                             dirtyb = Synchronizer.dirty[o][b] | {
      all p: Path | all a', b': Filesystem | {
       (relevant(a, b, p)&& Synchronizer.recon[o][a][b][p]= a'->b')
                           => (syncp(p, a', b', a, b, dirtya, dirtyb))
      }
    }
  }
}
```

This produces no counterexamples, giving us some assurance that there is no problem in the new description of reconciliation that was written. However, as demonstrated earlier when filesystems were found to not include files, it is important to add a sanity check to the model to be sure that the recon relation not only safely synchronizes filesystems, but also that it synchronizes all filesystems that it should:

```
assert reconGensAll {
  all o, a, b: Filesystem| all z: Path | {
    some Synchronizer.recon[o][a][b][z]
  }
}
```

However, this has counterexamples, revealing one interesting subtlety of modeling in Alloy. The statement was not true because in any given scope, there can be found filesystems that, in order to be synchronized, require more atoms than are allowed in the scope. Instead, we have to make a different kind of statement. We express that if we know that a synchronization exists for an original and two replicas in the scope, then the *recon* relation maps to some synchronization:

```
assert reconGensAll {
  all o, a, b: Filesystem | all z: Path | {
    let dirtya = Synchronizer.dirty[o][a],
        dirtyb = Synchronizer.dirty[o][b]  | {
          some a', b': Filesystem |
           {syncp(z, a', b', a, b, dirtya, dirtyb)}
             => some Synchronizer.recon[o][a][b][z]
    }
  }
}
```

However, this produces a counterexample involving filesystems that have no dirty paths. This is an artifact of modeling. When there are no dirty paths, the *recon* relation doesn't map the updated filesystems to anything despite the fact that what they should be mapped to is obvious. Since those cases are easily characterized, we modify the assertion slightly (*):

```
    assert reconGensAll {
      all o, a, b: Filesystem | all z: Path | {
        let dirtya = Synchronizer.dirty[o][a], dirtyb =
                              Synchronizer.dirty[o][b]  | {
(*)         (some dirtya && some dirtyb && some a', b': Filesystem |
                             {syncp(z, a', b', a, b, dirtya, dirtyb)})
                              => some Synchronizer.recon[o][a][b][z]
        }
      }
    }
```

Now there are no counterexamples and we have a model that we have some confidence safely synchronizes all filesystems that it should synchronize. As you can see, the process of creating this model was very error-prone, but with the aid of the Alloy analyzer it was possible to catch these problems without a lot of complicated reasoning on the part of the modeler.

Alloy models in full for this section are included in Appendix B and more detailed explanations of synchronization can be found in [2], a summary of which is present in the main body of this paper. A more detailed treatment of Alloy may be found in [6].

# Appendix B

# Alloy Models in Full

## B.1 Original Function Model of a Reconciliation Algorithm

```
module systems/file_system

open std/ord
open std/seq

sig Name {}

sig Path {
  names: Seq[Name]
}

sig Node {}

disj sig File extends Node {}

// Directories aren't explicitly named... instead, there
// are paths made up of names which may point to them
disj sig Filesystem extends Node {
  contents: Path ? -> ? Node
  } {
  // there is an empty path that maps directories to themselves
  some p: Path | EmptyPath(p) && this = p.contents
}

fact CanonicalPath {all a, b: Path| a.names = b.names => a = b}

fact CanonicalSequence {all a, b: Seq[Name] | a..SeqEquals(b) => a = b}

fact CanonicalNode {all a, b: Node | EquivNode(a, b) => a = b}

fact FileDirPartition {
  File + Filesystem = Node
}

// "Tree structure" not a statement about being acyclic...
// When you hit a directory on your path from your entries, that
// directory has entries consistent with yours. "Treeness" refers
// to the fact that things rooted under you "inherit" your mappings.
// Removal of g in File... disallows existence of files
fact TreeStructure {
  all f: Filesystem | all p, q: Path | {
    let g = p.f::contents | {
      (g in File && EmptyPath(q)) ||
          (q.g::contents = append(p, q).f::contents)
    }
  }
}

// If there is a path q.x then there is a path q
fact NoMissingPathBegin {
  all p: Path | {
    !EmptyPath(p) => {
      some q : Path | (PathPrefix(q, p) &&  (#PathLength(p) =
                                              #PathLength(q) + 1))
     }
  }
}

// Tails of paths are also paths
```

```
fact NoMissingPathEnd {
  all p: Path | {
    !EmptyPath(p) => {
      some q: Path | tail(p, q)
    }
  }
}

/***************  UPDATE DETECTION  ****************/

// Dirty can give you clean paths
// Unused paths are dirty often
// Safely estimates updates from f to g
// Changing to iff gives exact update detection
fun dirty (f, g: Filesystem): set Path {
  all p: Path | {
    p !in result => p.f::contents = p.g::contents
    // DirtyUpClosed
    else {all q: Path | PathPrefix(q, p) => q in result}
  }
}

/****************  RECONCILIATION  ****************/

// Synchronized filesystem after some designated path
fun afterPSynchronization (a, b, c, d: Node, dirtya, dirtyb: set Path,
                                                                   p: Path) {
  relevant(a, b, p)
  all pq: Path | {
    (PathPrefix(p, pq) && relevant(a, b, pq))
        => {
             pq !in dirtya => (pq.c::contents = pq.d::contents &&
                               pq.c::contents = pq.b::contents)
             pq !in dirtyb => (pq.c::contents = pq.d::contents &&
                               pq.c::contents = pq.a::contents)
             isDirAB(a, b, pq) => isDirAB(c, d, pq)
             (pq in dirtya && pq in dirtyb && !isDirAB(a, b, pq)) =>
```

```
                              (pq.c::contents = pq.a::contents &&
                               pq.d::contents = pq.b::contents)
           }
  }
}

// A synchronization after p such that nothing else changes
fun syncp (p: Path, c, d, a, b: Node, dirtya, dirtyb: set Path) {
  afterPSynchronization(a, b, c, d, dirtya, dirtyb, p)
  all q: Path | {
    incomparable(p, q) =>
             (q.c::contents = q.a::contents &&
              q.d::contents = q.b::contents)
    (PathPrefix(q, p) && isDirAB(a, b, q)) => isDirAB(c, d, q)
  }
}

// Takes pair of filesystems A, B and a path p and returns a pair of
// filesystems where subtrees rooted at p have been synchronized
fun recon(a, b: Node, p: Path, dirtya, dirtyb: set Path): Node->Node {
  one result
  (p !in dirtya && p !in dirtyb)
     => result = a->b
  else isDirAB(a, b, p)
     => reconhelper(a, result, b, dirtya, dirtyb, childrenAB(a, b, p))
  else p !in dirtya
     => {some abp: Node | {filesystemOverwrite(a, abp, b, p) &&
                                                      result = abp->b}}
  else p !in dirtyb
     => {some bap: Node | {filesystemOverwrite(b, bap, a, p) &&
                                                      result = a->bap}}
  else result = a->b
}

// Recursive portion of recon is commented out
fun reconhelper(a, b: Node, dirtya, dirtyb: set Path,
                                        childs: set Path): Node->Node {
```

```
  one result
  no childs => result = a->b
  else some q in childs | some c: Node->Node | {
    one q
//    let c = recon(a, b, q, dirtya, dirtyb) | {
//     reconhelper(firstNode(c), result, secondNode(c), dirtya, dirtyb,
//                                                          (childs - q))
//    }
  }
}

fun firstNode(c: Node -> Node) : Node {
  all a, b: Node | {
    c = a->b => result = a
  }
}

fun secondNode(c: Node -> Node) : Node {
  all a, b: Node | {
    c = a->b => result = b
  }
}

/******************  HELPERS  *******************/

fun EmptyPath (p: Path) {
  p.names..SeqIsEmpty()
}

// a <= b
fun PathPrefix (a, b: Path) {
  b.names..SeqStartsWith(a.names)
}

// a < b
fun PPathPrefix (a, b: Path) {
  PathPrefix(a, b) && #PathLength(a) != #PathLength(b)
```

```
}

fun tail (p: Path, q: Path) {
  p.names..SeqRest() = q.names
}

// Must be used in conjunction with #
fun PathLength (p: Path): set SeqIdx  {
  result = SeqInds(p.names)
}

// All paths at some node
fun Paths(f: Filesystem): set Path {
  result = Node.~(f.contents)
}

// Path Append
fun append (a, b: Path): Path {
  result = {r: Path | {
    #PathLength(r) = #PathLength(a) + #PathLength(b)
    PathPrefix(a, r)
    all i: r.names..SeqInds() | {
      #OrdPrevs(i) >= #PathLength(a) =>
         some k: SeqIdx | {
           #OrdPrevs(k) + #PathLength(a)= #OrdPrevs(i)
           b.names..SeqAt(k) = r.names..SeqAt(i)
         }
    }
  }}
}

// Returns paths for children
fun children (f: Filesystem, p: Path): set Path {
   result = {q: Path | {
     q in Node.~f::contents
     PathPrefix(p, q)
     #PathLength(p) + 1  = #PathLength(q)
   }}
}

fun childrenAB (f, g: Filesystem, p: Path): set Path {
  result = (children(f, p) + children(g, p))
}

fun isDir (f: Filesystem, p: Path) {
  some (Filesystem & p.f::contents)
}

fun isDirAB (f, g: Filesystem, p:Path) {
  isDir(f, p) && isDir(g, p)
}

// Equivalent filesystems map paths to same files and directory structure
// On empty inputs, returns true
fun EquivNode (f, g: Node) {
  (f = g) ||
  (f in Filesystem && g in Filesystem && Paths(f) = Paths(g) &&
    all p: Path | all e: File | {
      p->e in f.contents <=> p->e in g.contents
    }
  )
}

// Path p is relevant if all its ancestors refer to dirs in both dirs
// Note restriction that a, b are Filesystems and not just nodes...
// Otherwise, synchronization is often trivially true
fun relevant (a, b: Filesystem, p: Path) {
  all q: Path | {
    PPathPrefix(q, p) => isDirAB(a, b, q)
  }
}

// Two paths are on "different branches"
fun incomparable (p, q: Path) {
  !PathPrefix(p, q) && !PathPrefix(q, p)
}

// Replace subtree at p in T with S's
fun filesystemOverwrite (t, s: Node, p: Path): Node {
  all q: Path | {
     PathPrefix(p, q) => q.result::contents = q.s::contents
     else incomparable(p, q) => q.result::contents = q.t::contents
     else children(result, q) + p = children(t, q) + p
  }
}

/***************  CHECKING ***************/

// Recon satisfies the requirements for synchronization
// Works if you exclude cases that would use recursion
assert Soundness {
  all a, b, o: Filesystem | {
    let dirtya = dirty(o, a), dirtyb = dirty(o, b) | {
      all z: Path | all c, d: Filesystem | {
        (relevant(a, b, z) && recon(a, c->d, b, z, dirtya, dirtyb))
                                =>(syncp(z, c, d, a, b, dirtya, dirtyb))
      }
    }
  }
}

assert reconGensAll {
  all o, a, b: Filesystem | all z: Path | all dirtya, dirtyb: set Path | {
    (dirty(o, dirtya, a) && dirty(o, dirtyb, b) && some c, d: Filesystem |
       {syncp(z, c, d, a, b, dirtya, dirtyb)})
         => some c, d: Filesystem | {recon(a, c->d, b, z, dirtya, dirtyb)}
  }
}

// Only one synchronization of a pair of filesystems w.r.t.
// dirty predicates
```

```
assert Uniqueness {
  all a, b, o: Filesystem | all dirtya, dirtyb: set Path | {
     all p: Path | all c1, d1, c2, d2: Filesystem | {
      (dirty(o, dirtya, a) && dirty(o, dirtyb, b) && relevant(a, b, p) &&
               afterPSynchronization(a, b, c1, d1, dirtya, dirtyb, p) &&
               afterPSynchronization(a, b, c2, d2, dirtya, dirtyb, p))
                    =>   (EquivNode(p.c1::contents, p.c2::contents) &&
                           EquivNode(p.d1::contents, p.d2::contents))
     }
  }
}

// If a path is dirty in either updated replica then the path
// maps to equivalent nodes in both
assert NonDirtyAreSame {
  all a, b, o: Filesystem | all p: Path | all dirtya, dirtyb: set Path |{
    (dirty(o, dirtya, a) && dirty(o, dirtyb, b) && p !in dirtya &&
                                                         p !in dirtyb)
            => EquivNode(p.a::contents, p.b::contents)
  }
}

// True as long as don't get rid of NoMissingPath facts and the
// restriction on filesystem signature at once
assert NoCycles  {
  all f: Filesystem | no p: Path | {
    !EmptyPath(p) && p->f in f.contents
  }
}
```

## B.2 Manually Unrolled Function Model of Reconciliation Algorithm

```
module systems/file_system

open std/ord
open std/seq

sig Name {}

sig Path {
  path: Seq[Name]
}

sig Node {}

disj sig File extends Node {}

// Directories aren't explicitly named... instead, there
// are paths made up of names which may point to them
disj sig Filesystem extends Node {
  contents: Path ? -> ? Node
  } {
  // there is an empty path that maps directories to themselves
  some p: Path | EmptyPath(p) && this = p.contents
}

fact CanonicalPath {all a, b: Path| a.names = b.names => a = b}

fact CanonicalSequence {all a, b: Seq[Name] | a..SeqEquals(b) => a = b}

fact CanonicalNode {all a, b: Node | EquivNode(a, b) => a = b}

fact FileDirPartition {
  File + Filesystem = Node
}
```

```
// "Tree structure" not a statement about being acyclic...
// When you hit a directory on your path from your entries, that
// directory has entries consistent with yours. "Treeness" refers
// to the fact that things rooted under you "inherit" your mappings.
// Removal of g in File... disallows existence of files
fact TreeStructure {
  all f: Filesystem | all p, q: Path | {
    let g = p.f::contents | {
      (g in File && EmptyPath(q)) ||
          (q.g::contents = append(p, q).f::contents)
    }
  }
}

// If there is a path q.x then there is a path q
fact NoMissingPathBegin {
  all p: Path | {
    !EmptyPath(p) => {
      some q : Path | (PPathPrefix(q, p) &&  (#PathLength(p) =
                                               #PathLength(q) + 1))
     }
  }
}

// Tails of paths are also paths
fact NoMissingPathEnd {
  all p: Path | {
    !EmptyPath(p) => {
      some q: Path | tail(p, q)
    }
  }
}

/***************  UPDATE DETECTION  ****************/

// Dirty can give you clean paths
// Unused paths are dirty often
// Safely estimates updates from f to g
// Changing to iff gives exact update detection
fun dirty (f, g: Filesystem): set Path {
  all p: Path | {
    p !in result => p.f::contents = p.g::contents
    // DirtyUpClosed
    else {all q: Path | PathPrefix(q, p) => q in result}
  }
}

/****************  RECONCILIATION  ****************/

// Synchronized filesystem after some designated path
fun afterPSynchronization (a, b, c, d: Node, dirtya, dirtyb: set Path,
                                                              p: Path) {
  relevant(a, b, p)
  all pq: Path | {
    (PathPrefix(p, pq) && relevant(a, b, pq))
        => {
             pq !in dirtya => (pq.c::contents = pq.d::contents &&
                               pq.c::contents = pq.b::contents)
             pq !in dirtyb => (pq.c::contents = pq.d::contents &&
                               pq.c::contents = pq.a::contents)
             isDirAB(a, b, pq) => isDirAB(c, d, pq)
             (pq in dirtya && pq in dirtyb && !isDirAB(a, b, pq)) =>
                              (pq.c::contents = pq.a::contents &&
                               pq.d::contents = pq.b::contents)
           }
  }
}

// A synchronization after p such that nothing else changes
fun syncp (p: Path, c, d, a, b: Node, dirtya, dirtyb: set Path) {
  afterPSynchronization(a, b, c, d, dirtya, dirtyb, p)
  all q: Path | {
    incomparable(p, q) =>
             (q.c::contents = q.a::contents &&
              q.d::contents = q.b::contents)
    (PathPrefix(q, p) && isDirAB(a, b, q)) => isDirAB(c, d, q)
  }
}

fun recon(a, b, c, d: Node, p: Path, dirtya, dirtyb: set Path) {
  all q: Path | {
    incomparable(p, q) => (q.c::contents = q.a::contents &&
                           q.d::contents = q.b::contents)
    (q !in Paths(a) && q !in Paths(b)) =>
                     (q !in Paths(c) && q !in Paths(d))
    (q in Paths(a) && q in Paths(b)) =>
                     (q in Paths(c) && q in Paths(d))
    isDirAB(a, b, q) => isDirAB(c, d, q)
  }
  (p !in Paths(a) + Paths(b))
     => (c = a && d = b)
  else (p !in dirtya + dirtyb)
     => (c = a && d = b)
  else isDirAB(a, b, p)
     => {no childrenAB(a, b, p) => (c = a && d = b)
         else all q in childrenAB(a, b, p) | {
              reconhelper(a, b, c, d, q, dirtya, dirtyb)
         }}
  else p !in dirtya
     => (d = b && filesystemOverwrite(a, c, b, p))
  else p !in dirtyb
     => (c = a && filesystemOverwrite(b, d, a, p))
  else (c = a && d = b)
}

fun reconhelper(a, b, c, d: Node, p: Path, dirtya, dirtyb: set Path) {
  (p !in dirtya + dirtyb)
     => (p.c::contents = p.a::contents && p.d::contents = p.b::contents)
  else isDirAB(a, b, p)
     => {no childrenAB(a, b, p) => (p.c::contents = p.a::contents &&
                                    p.d::contents = p.b::contents)
```

```
          else all q in childrenAB(a, b, p) | {
               reconhelper2(a, b, c, d, q, dirtya, dirtyb)
          }}
  else p !in dirtya
     => (p.d::contents = p.b::contents && filesystemOverwrite(a, c, b, p))
  else p !in dirtyb
     => (p.c::contents = p.a::contents && filesystemOverwrite(b, d, a, p))
  else (p.c::contents = p.a::contents && p.d::contents = p.b::contents)
}

...

fun reconhelper4(a, b, c, d: Node, p: Path, dirtya, dirtyb: set Path) {
  (p !in dirtya + dirtyb)
     => (p.c::contents = p.a::contents && p.d::contents = p.b::contents)
  else isDirAB(a, b, p)
     => {no childrenAB(a, b, p) => (p.c::contents = p.a::contents &&
                                    p.d::contents = p.b::contents)
          else all q in childrenAB(a, b, p) | {
               reconhelper5(a, b, c, d, q, dirtya, dirtyb)
          }}
  else p !in dirtya
     => (p.d::contents = p.b::contents && filesystemOverwrite(a, c, b, p))
  else p !in dirtyb
     => (p.c::contents = p.a::contents && filesystemOverwrite(b, d, a, p))
  else (p.c::contents = p.a::contents && p.d::contents = p.b::contents)
}

// Terminating
fun reconhelper5(a, b, c, d: Node, p: Path, dirtya, dirtyb: set Path) {
  (p !in dirtya + dirtyb)
     => (p.c::contents = p.a::contents && p.d::contents = p.b::contents)
  else p !in dirtya
     => (p.d::contents = p.b::contents && filesystemOverwrite(a, c, b, p))
  else p !in dirtyb
     => (p.c::contents = p.a::contents && filesystemOverwrite(b, d, a, p))
  else (p.c::contents = p.a::contents && p.d::contents = p.b::contents)
```

```
}

/******************  HELPERS  *******************/

fun EmptyPath (p: Path) {
  p.names..SeqIsEmpty()
}

// a <= b
fun PathPrefix (a, b: Path) {
  b.names..SeqStartsWith(a.names)
}

// a < b
fun PPathPrefix (a, b: Path) {
  PathPrefix(a, b) && #PathLength(a) != #PathLength(b)
}

fun tail (p: Path, q: Path) {
  p.names..SeqRest() = q.names
}

// Must be used in conjunction with #
fun PathLength (p: Path): set SeqIdx  {
  result = SeqInds(p.names)
}

// All paths at some node
fun Paths(f: Filesystem): set Path {
  result = Node.~(f.contents)
}

// Path Append
fun append (a, b: Path): Path {
  result = {r: Path | {
    #PathLength(r) = #PathLength(a) + #PathLength(b)
    PathPrefix(a, r)
    all i: r.names..SeqInds() | {
      #OrdPrevs(i) >= #PathLength(a) =>
         some k: SeqIdx | {
           #OrdPrevs(k) + #PathLength(a)= #OrdPrevs(i)
           b.names..SeqAt(k) = r.names..SeqAt(i)
         }
    }
  }}
}

// Returns paths for children
fun children (f: Filesystem, p: Path): set Path {
   result = {q: Path | {
     q in Node.~f::contents
     PathPrefix(p, q)
     #PathLength(p) + 1  = #PathLength(q)
   }}
}

fun childrenAB (f, g: Filesystem, p: Path): set Path {
  result = (children(f, p) + children(g, p))
}

fun isDir (f: Filesystem, p: Path) {
  some (Filesystem & p.f::contents)
}

fun isDirAB (f, g: Filesystem, p:Path) {
  isDir(f, p) && isDir(g, p)
}

// Equivalent filesystems map paths to same files and directory structure
// On empty inputs, returns true
fun EquivNode (f, g: Node) {
  (f = g) ||
  (f in Filesystem && g in Filesystem && Paths(f) = Paths(g) &&
    all p: Path | all e: File | {
      p->e in f.contents <=> p->e in g.contents
    }
  )
}

// Path p is relevant if all its ancestors refer to dirs in both dirs
// Note restriction that a, b are Filesystems and not just nodes...
// Otherwise, synchronization is often trivially true
fun relevant (a, b: Filesystem, p: Path) {
  all q: Path | {
    PPathPrefix(q, p) => isDirAB(a, b, q)
  }
}

// Two paths are on "different branches"
fun incomparable (p, q: Path) {
  !PathPrefix(p, q) && !PathPrefix(q, p)
}

// Replace subtree at p in T with S's
fun filesystemOverwrite (t, s: Node, p: Path): Node {
  all q: Path | {
    PathPrefix(p, q) => q.result::contents = q.s::contents
  }
}

/***************  CHECKING ***************/

// Recon satisfies the requirements for synchronization
assert Soundness {
  all a, b, o: Filesystem | all dirtya, dirtyb: set Path | {
      all z: Path | all c, d: Filesystem | {
        (dirty(o, dirtya, a) && dirty(o, dirtyb, b) &&
          relevant(a, b, z) && recon(a, b, c, d, z, dirtya, dirtyb)) =>
                                   (syncp(z, c, d, a, b, dirtya, dirtyb))
      }
  }
```

```
}

assert reconGensAll {
  all o, a, b: Filesystem | all z: Path | all dirtya, dirtyb: set Path | {
    (dirty(o, dirtya, a) && dirty(o, dirtyb, b) && some c, d: Filesystem |
      {syncp(z, c, d, a, b, dirtya, dirtyb)})
        => some c, d: Filesystem | {recon(a, b, c, d, z, dirtya, dirtyb)}
  }
}

// Only one synchronization of a pair of filesystems w.r.t.
// dirty predicates
assert Uniqueness {
  all a, b, o: Filesystem | all dirtya, dirtyb: set Path | {
     all p: Path | all c1, d1, c2, d2: Filesystem | {
      (dirty(o, dirtya, a) && dirty(o, dirtyb, b) && relevant(a, b, p) &&
                afterPSynchronization(a, b, c1, d1, dirtya, dirtyb, p) &&
                afterPSynchronization(a, b, c2, d2, dirtya, dirtyb, p))
                           =>      (p.c1::contents = p.c2::contents &&
                                    p.d1::contents = p.d2::contents)
     }
  }
}

// If a path isn't dirty in either updated replica then the path
// maps to equivalent nodes in both
assert NonDirtyAreSame {
  all a, b, o: Filesystem | all p: Path | all dirtya, dirtyb: set Path |{
    (dirty(o, dirtya, a) && dirty(o, dirtyb, b) && p !in dirtya &&
                                                   p !in dirtyb)
          => p.a::contents = p.b::contents
  }
}

// True as long as don't get rid of NoMissingPath fact and the
// restriction on filesystem signature at once
assert NoCycles  {
  all f: Filesystem | no p: Path | {
    !EmptyPath(p) && p->f in f.contents
  }
}
```

## B.3 Relational Model of Reconciliation Algorithm

```
module systems/file_system

open std/ord
open std/seq

sig Name {}

sig Path {
  path: Seq[Name]
}

sig Node {}

disj sig File extends Node {}

// Directories aren't explicitly named... instead, there
// are paths made up of names which may point to them
disj sig Filesystem extends Node {
  contents: Path ? -> ? Node
  } {
  // there is an empty path that maps directories to themselves
  some p: Path | EmptyPath(p) && this = p.contents
}

// Recon maps original, 2 updates, and a path to new filesystems
// Dirty maps pairs of old and updated filesystems to paths
static sig Synchronizer {
  recon: Node -> Node -> Node -> Path -> !(Node->Node),
  dirty : Node -> Node -> Path
}

fact CanonicalPath {all a, b: Path| a.names = b.names => a = b}

fact CanonicalSequence {all a, b: Seq[Name] | a..SeqEquals(b) => a = b}

fact CanonicalNode {all a, b: Node | EquivNode(a, b) => a = b}
fact FileDirPartition {
  File + Filesystem = Node
}

// "Tree structure" not a statement about being acyclic...
// When you hit a directory on your path from your entries, that
// directory has entries consistent with yours. "Treeness" refers
// to the fact that things rooted under you "inherit" your mappings.
// Removal of g in File... disallows existence of files
fact TreeStructure {
  all f: Filesystem | all p, q: Path | {
    let g = p.f::contents | {
      (g in File && EmptyPath(q)) ||
          (q.g::contents = append(p, q).f::contents)
    }
  }
}

// If there is a path q.x then there is a path q
fact NoMissingPathBegin {
  all p: Path | {
    !EmptyPath(p) => {
      some q : Path | (PathPrefix(q, p) &&  (#PathLength(p) =
                                                 #PathLength(q) + 1))
     }
  }
}

// Tails of paths are also paths
fact NoMissingPathEnd {
  all p: Path | {
    !EmptyPath(p) => {
      some q: Path | tail(p, q)
    }
  }
}
```

```
/***************  UPDATE DETECTION  ****************/

// Dirty can give you clean paths
// Unused paths are dirty often
// Safely estimates updates from f to g
// Changing to iff gives exact update detection

fact DirtyBehaviour {
  all f, g: Filesystem | all p: Path | {
    p !in Synchronizer.dirty[f][g] => p.f::contents = p.g::contents
  }
}

fact DirtyUpClosed {
  all f, g: Filesystem | all p, q: Path | {
    (PathPrefix(p, q) && q in Synchronizer.dirty[f][g]) =>
                               p in Synchronizer.dirty[f][g]
  }
}

/****************  RECONCILIATION  ****************/

// Synchronized filesystem after some designated path
fun afterPSynchronization (a, b, c, d: Node, dirtya, dirtyb: set Path,
                                                               p: Path) {
  relevant(a, b, p)
  all pq: Path | {
    (PathPrefix(p, pq) && relevant(a, b, pq))
        => {
             pq !in dirtya => (pq.c::contents = pq.d::contents &&
                               pq.c::contents = pq.b::contents)
             pq !in dirtyb => (pq.c::contents = pq.d::contents &&
                               pq.c::contents = pq.a::contents)
             isDirAB(a, b, pq) => isDirAB(c, d, pq)
             (pq in dirtya && pq in dirtyb && !isDirAB(a, b, pq)) =>
                             (pq.c::contents = pq.a::contents &&
                              pq.d::contents = pq.b::contents)
          }
  }
}

// A synchronization after p such that nothing else changes
fun syncp (p: Path, c, d, a, b: Node, dirtya, dirtyb: set Path) {
  afterPSynchronization(a, b, c, d, dirtya, dirtyb, p)
  all q: Path | {
    incomparable(p, q) =>
             (q.c::contents = q.a::contents &&
              q.d::contents = q.b::contents)
    (PathPrefix(q, p) && isDirAB(a, b, q)) => isDirAB(c, d, q)
  }
}

// Constraining behaviour of recon mapping
fact reconFacts {
  all o, a, b: Node | all p: Path | some abp, bap: Node |{
    let dirtya = Synchronizer.dirty[o][a],
          dirtyb = Synchronizer.dirty[o][b] | {
      // p didn't change in either filesystem, so don't change anything
      (p !in dirtya + dirtyb)
        => Synchronizer.recon[o][a][b][p] = a->b
      // recursive portion... recon of a dirty directory has property
      // that it has same contents as the recon of subdirectories
      else isDirAB(a, b, p)
        => {
          no childrenAB(a, b, p) => Synchronizer.recon[o][a][b][p] = a->b
          all q: childrenAB(a, b, p) | {
            let c = Node.~(Synchronizer::recon[o][a][b][p]),
                d = Synchronizer.recon[o][a][b][p][c],
                e = Node.~(Synchronizer.recon[o][a][b][q]),
                f = Synchronizer.recon[o][a][b][q][e] | {
                  (children(c, p)+ children(d, p)) in childrenAB(a, b, p)
                    all s: Path | {PathPrefix(s, q) =>
                                  ((s in Paths(c) <=> s in Paths(e)) &&
                                (s in Paths(d) <=> s in Paths(f)))}
                    (q.c::contents = q.e::contents &&
                       q.d::contents = q.f::contents)
              }
           }
         }
      // otherwise, if only dirty in one filesystem, overwrite unchanged
      else p !in dirtya
        => filesystemOverwrite(a, abp, b, p) &&
                    Synchronizer.recon[o][a][b][p] = abp->b
      else p !in dirtyb
        => filesystemOverwrite(b, bap, a, p) &&
                    Synchronizer.recon[o][a][b][p] = a->bap
      // else it's a conflicting update so don't change anything
      else Synchronizer.recon[o][a][b][p] = a->b
    }
  }
}

/******************  HELPERS  *******************/

fun EmptyPath (p: Path) {
  p.names..SeqIsEmpty()
}

// a <= b
fun PathPrefix (a, b: Path) {
  b.names..SeqStartsWith(a.names)
}

// a < b
fun PPathPrefix (a, b: Path) {
  PathPrefix(a, b) && #PathLength(a) != #PathLength(b)
}

fun tail (p: Path, q: Path) {
  p.names..SeqRest() = q.names
```

```
}

// Must be used in conjunction with #
fun PathLength (p: Path): set SeqIdx  {
  result = SeqInds(p.names)
}

// All paths at some node
fun Paths(f: Filesystem): set Path {
  result = Node.~(f.contents)
}

// Path Append
fun append (a, b: Path): Path {
  result = {r: Path | {
   #PathLength(r) = #PathLength(a) + #PathLength(b)
   PathPrefix(a, r)
   all i: r.names..SeqInds() | {
     #OrdPrevs(i) >= #PathLength(a) =>
        some k: SeqIdx | {
          #OrdPrevs(k) + #PathLength(a)= #OrdPrevs(i)
          b.names..SeqAt(k) = r.names..SeqAt(i)
        }
   }
  }}
}

// Returns paths for children
fun children (f: Filesystem, p: Path): set Path {
  result = {q: Path | {
       q in Node.~f::contents
       PathPrefix(p, q)
       #PathLength(p) + 1  = #PathLength(q)
       }
  }
}
```

```
fun childrenAB (f, g: Filesystem, p: Path): set Path {
  result = (children(f, p) + children(g, p))
}

fun isDir (f: Filesystem, p: Path) {
  some (Filesystem & p.f::contents)
}

fun isDirAB (f, g: Filesystem, p:Path) {
  isDir(f, p) && isDir(g, p)
}

// Equivalent filesystems map paths to same files and directory structure
// On empty inputs, returns true
fun EquivNode (f, g: Node) {
  (f = g) ||
  (f in Filesystem && g in Filesystem && Paths(f) = Paths(g) &&
    all p: Path | all e: File | {
      p->e in f.contents <=> p->e in g.contents
    }
  )
}

// Path p is relevant if all its ancestors refer to dirs in both dirs
// Note restriction that a, b are Filesystems and not just nodes...
// Otherwise, synchronization is often trivially true
fun relevant (a, b: Filesystem, p: Path) {
  all q: Path | {
    PPathPrefix(q, p) => isDirAB(a, b, q)
  }
}

// Two paths are on "different branches"
fun incomparable (p, q: Path) {
  !PathPrefix(p, q) && !PathPrefix(q, p)
}
```

```
// Replace subtree at p in T with S's
fun filesystemOverwrite (t, s: Node, p: Path): Node {
  p !in (Node.~(s::contents) + Node.~(t::contents)) => result = t
  else all q: Path | {
     PathPrefix(p, q) => q.result::contents = q.s::contents
     else incomparable(p, q) => q.result::contents = q.t::contents
     else children(result, q) + p = children(t, q) + p
  }
}

/***************  CHECKING **************/

// Recon doesn't miss cases
assert reconGensAll {
  all o, a, b: Filesystem | all z: Path | {
    let dirtya = Synchronizer.dirty[o][a],
           dirtyb = Synchronizer.dirty[o][b] | {
      (some dirtya && some dirtyb && some c, d: Filesystem |
        {syncp(z, c, d, a, b, dirtya, dirtyb)})
           => some Synchronizer.recon[o][a][b][z]
    }
  }
}

// Recon satisfies the requirements for synchronization
assert Soundness {
  all a, b, o: Filesystem | {
   let dirtya = Synchronizer.dirty[o][a],
         dirtyb = Synchronizer.dirty[o][b] | {
      all p: Path | all c, d: Filesystem | {
        (relevant(a, b, p) && Synchronizer.recon[o][a][b][p] = c->d) =>
                                   (syncp(p, c, d, a, b, dirtya, dirtyb))
      }
    }
  }
}
```

```
// Only one synchronization of a pair of filesystems w.r.t.
// dirty predicates
assert Uniqueness {
  all a, b, o: Filesystem | {
    let dirtya = Synchronizer.dirty[o][a],
           dirtyb = Synchronizer.dirty[o][b] | {
     all p: Path | all c1, d1, c2, d2: Filesystem | {
      (relevant(a, b, p) &&
          afterPSynchronization(a, b, c1, d1, dirtya, dirtyb, p) &&
           afterPSynchronization(a, b, c2, d2, dirtya, dirtyb, p))
                              => (p.c1::contents = p.c2::contents &&
                                   p.d1::contents = p.d2::contents)
     }
   }
  }
}

// If a path is dirty in either updated replica then the path
// maps to equivalent nodes in both
assert NonDirtyAreSame {
  all a, b, o: Filesystem | all p: Path | {
    (p !in Synchronizer.dirty[o][a] && p !in Synchronizer.dirty[o][b])
             => p.a::contents = p.b::contents
  }
}

// True as long as don't get rid of NoMissingPath fact and the
// restriction on filesystem signature at once
assert NoCycles  {
  all f: Filesystem | no p: Path | {
    !EmptyPath(p) && p->f in f.contents
  }
}
```

## B.4 Models of Unison and Briefcase Specifications

```
module systems/file_system

open std/ord
open std/seq

sig Name {}

sig Path {
  path: Seq[Name]
}

sig Node {}

disj sig File extends Node {}

// Directories aren't explicitly named... instead, there
// are paths made up of names which may point to them
disj sig Filesystem extends Node {
  contents: Path ? -> ? Node
  } {
  // there is an empty path that maps directories to themselves
  some p: Path | EmptyPath(p) && this = p.contents
}

fact CanonicalPath {all a, b: Path| a.names = b.names => a = b}

fact CanonicalSequence {all a, b: Seq[Name] | a..SeqEquals(b) => a = b}

fact CanonicalNode {all a, b: Node| EquivNode(a, b) => a = b}

fact FileDirPartition {
  File + Filesystem = Node
}

// "Tree structure" not a statement about being acyclic...
// When you hit a directory on your path from your entries, that
// directory has entries consistent with yours. "Treeness" refers
// to the fact that things rooted under you "inherit" your mappings.
// Removal of g in File... disallows existence of files
fact TreeStructure {
  all f: Filesystem | all p, q: Path | {
    let g = p.f::contents | {
      (g in File && EmptyPath(q)) ||
          (q.g::contents = append(p, q).f::contents)
    }
  }
}

// If there is a path q.x then there is a path q
fact NoMissingPathBegin {
  all p: Path | {
    !EmptyPath(p) => {
      some q : Path | (PathPrefix(q, p) &&  (#PathLength(p) =
                                                  #PathLength(q) + 1))
     }
  }
}

// Tails of paths are also paths
fact NoMissingPathEnd {
  all p: Path | {
    !EmptyPath(p) => {
      some q: Path | tail(p, q)
    }
  }
}

/***************  UPDATE DETECTION  ****************/

fun dirty (f, g: Filesystem): set Path {
  all p: Path | {
    p !in result <=> p.f::contents = p.g::contents
  }
```

```
}

fun conflict(o, a, b: Node) {
  !(samesort(a, b) || o = a || o = b)
}

/***************** Unison recon *****************/

fun relevant (a, b: Filesystem, p: Path) {
  all q: Path | {
    PPathPrefix(q, p) => isDirAB(a, b, q)
  }
}

fun PreserveLocalChanges(o, a, b, c, d: Node) {
  !samesort(o, a) => samesort(c, a)
  !samesort(o, b) => samesort(d, b)
}

fun PropagateOnlyUserChanges(o, a, b, c, d: Node) {
  !samesort(a, c) => samesort(b, c)
  !samesort(b, d) => samesort(a, d)
}

fun StopAtConflicts(o, a, b, c, d: Node) {
  conflict(o, a, b) => (c = a && d = b)
}

fun safe(o, a, b, c, d: Node) {
  all p: Path | {
    let o' = p.o::contents, a' = p.a::contents,
        b' = p.b::contents, c' = p.c::contents,
                          d' = p.d::contents | {
          PreserveLocalChanges(o', a', b', c', d')
          PropagateOnlyUserChanges(o', a', b', c', d')
          StopAtConflicts(o', a', b', c', d')
    }
  }
}

fun maximal(o, a, b, c, d: Node) {
  all c', d': Node | {
    safe(o, a, b, c', d') =>
      all p: Path | {
        samesort(p.c'::contents, p.d'::contents)
           => samesort(p.c::contents, p.d::contents)
      }
  }
}

// c and d are the synchronized versions of a and b w.r.t. dirty sets
fun synchronization (a, b, c, d: Node, dirtya, dirtyb: set Path) {
  all j: Path | {
    relevant(a, b, j) => {
      j !in dirtya =>
                 (j.c::contents = j.d::contents &&
                  j.c::contents = j.b::contents)
      j !in dirtyb =>
                 (j.c::contents = j.d::contents &&
                  j.c::contents = j.a::contents)
      isDirAB(a, b, j) => isDirAB(c, d, j)
      (j in dirtya && j in dirtyb && !isDirAB(a, b, j)) =>
                 (j.c::contents = j.a::contents &&
                  j.d::contents = j.b::contents)
    }
  }
}

// Nonconflicting updates are propagated
fun unisonRecon(o, a, b, c, d: Node) {
  all p: Path | {
    (samesort(p.o::contents, p.a::contents) &&
       samesort(p.o::contents, p.b::contents))
      => (samesort(p.a::contents, p.c::contents) &&
          samesort(p.b::contents, p.d::contents))
    else (samesort(p.o::contents, p.a::contents) &&
          {all q: Path | PathPrefix(q, p)
              => !conflict(q.o::contents, q.a::contents, q.b::contents)})
                    => (samesort(p.b::contents, p.c::contents) &&
                        samesort(p.b::contents, p.d::contents))
    else (samesort(p.o::contents, p.b::contents) &&
          {all q: Path | PathPrefix(q, p)
              => !conflict(q.o::contents, q.a::contents, q.b::contents)})
                    => (samesort(p.a::contents, p.c::contents) &&
                        samesort(p.a::contents, p.d::contents))
    else (samesort(p.a::contents, p.c::contents) &&
          samesort(p.b::contents, p.d::contents))
  }
}

// Nonconflicting updates are propagated
// Not maximal in the sense that it does not synchronize all it should
fun BadUnisonRecon(o, a, b, c, d: Node) {
  all p: Path | {
    (samesort(p.o::contents, p.a::contents) &&
                          samesort(p.o::contents, p.b::contents))
      => (samesort(p.a::contents, p.c::contents) &&
                              samesort(p.b::contents, p.d::contents))
    else (samesort(p.o::contents, p.a::contents) &&
                 !conflict(p.o::contents, p.a::contents, p.b::contents))
      => (samesort(p.b::contents, p.c::contents) &&
                              samesort(p.b::contents, p.d::contents))
    else (samesort(p.o::contents, p.b::contents) &&
                 !conflict(p.o::contents, p.a::contents, p.b::contents))
      => (samesort(p.a::contents, p.c::contents) &&
                              samesort(p.a::contents, p.d::contents))
    else (samesort(p.a::contents, p.c::contents) &&
                              samesort(p.b::contents, p.d::contents))
  }
}
```

```
/******************  HELPERS  ********************/

fun EmptyPath (p: Path) {
  p.names..SeqIsEmpty()
}

// a <= b
fun PathPrefix (a, b: Path) {
  b.names..SeqStartsWith(a.names)
}

// a < b
fun PPathPrefix (a, b: Path) {
  PathPrefix(a, b) && #PathLength(a) != #PathLength(b)
}

fun tail (p: Path, q: Path) {
  p.names..SeqRest() = q.names
}

// Must be used in conjunction with #
fun PathLength (p: Path): set SeqIdx  {
  result = SeqInds(p.names)
}

// All paths at some node
fun Paths(f: Filesystem): set Path {
  result = Node.~(f.contents)
}

// Path Append
fun append (a, b: Path): Path {
  result = {r: Path | {
    #PathLength(r) = #PathLength(a) + #PathLength(b)
    PathPrefix(a, r)
    all i: r.names..SeqInds() | {
      #OrdPrevs(i) >= #PathLength(a) =>
```

```
          some k: SeqIdx | {
            #OrdPrevs(k) + #PathLength(a)= #OrdPrevs(i)
            b.names..SeqAt(k) = r.names..SeqAt(i)
          }
    }
  }}
}

fun isDir (f: Filesystem, p: Path) {
  some (Filesystem & p.f::contents)
}

fun isDirAB (f, g: Filesystem, p:Path) {
  isDir(f, p) && isDir(g, p)
}

// Equivalent filesystems map paths to same files and directory structure
// On empty inputs, returns true
fun EquivNode (f, g: Node) {
  (f = g) ||
  (f in Filesystem && g in Filesystem && Paths(f) = Paths(g) &&
    all p: Path | all e: File | {
      p->e in f.contents <=> p->e in g.contents
    }
  )
}

fun samesort(a, b: Node) {
  a = b || (some (a & Filesystem) && some (b & Filesystem))
}

/************CHECKING********************************/

assert checkUnisonSafe {
  all o, a, b, c, d: Node | {
    unisonRecon(o, a, b, c, d) =>
     (safe(o, a, b, c, d) &&
      maximal(o, a, b, c, d))
  }
}

// Not true due to artifact of modeling
assert UnisonGensAll {
  all o, a, b, c, d: Filesystem | {
   (safe(o, a, b, c, d) &&
     maximal(o, a, b, c, d)) =>
       unisonRecon(o, a, b, c, d)
  }
}

// Checks what UnisonGensAll was intended to check
assert unisonMaximal {
  all o, a, b, c, d: Node, dirtya, dirtyb: set Path | {
    (dirty(o, dirtya, a) && dirty(o, dirtyb, b) &&
                          synchronization(a, b, c, d, dirtya, dirtyb))
         => unisonRecon(o, a, b, c, d)
  }
}

// If Unison reconciles, then recon does as well
assert unisonReconisRecon {
  all o, a, b, c, d: Filesystem | all z: Path | {
    (EmptyPath(z) && unisonRecon(o, a, b, c, d)) => {
      some dirtya, dirtyb: set Path | {
        recon(a, b, c, d, z, dirtya, dirtyb)
        dirty(o, dirtya, a)
        dirty(o, dirtyb, b)

    }
  }
}

assert laziness {
  all o, a, b: Node | safe(o, a, b, a, b)
```

```
}

assert mirroring {
  all o, a: Node | maximal(o, a, o, a, a)
}

assert briefcaseMirroring {
  all o, a: Node | unisonRecon(o, a, o, a, a)
}

assert briefcaseSymmetricMirroring {
  all o, a: Node | unisonRecon(o, o, a, a, a)
}

assert briefcase {
  all o, a, o', a', b, b', o", a", o"': Node | {
    /*a is synchronized first*/
    {unisonRecon(o, a, o, a', o')
     /*b is synchronized with the updated network version*/
     unisonRecon(o, b, o', b', o")
     /*a is synchronized again with network to get b's updates*/
     unisonRecon(o', a', o", a", o"')}
          /*This is the same as doing unisonRecon to begin with so*/
          /*this gives correctness of interpreted Briefcase spec*/
         => unisonRecon(o, a, b, a", b')
  }
}

assert maximalUnique {
  all o, a, b, c, d, c', d': Node | {
    (safe(o, a, b, c, d) && safe(o, a, b, c', d') &&
     maximal(o, a, b, c, d) && maximal(o, a, b, c', d')) =>
      all p: Path | {
        samesort(p.c::contents, p.c'::contents)
      }
  }
}
```

```
assert noConflictSuccess {
  all o, a, b: Node | {
    all p: Path | {
      !conflict(p.o::contents, p.a::contents, p.b::contents)
    } => some c: Node | {maximal(o, a, b, c, c)}
  }
}
```

4935-51