# Models and Software Model Checking of a Distributed File Replication System[1]

Nikolaj Bjørner

Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
{nbjorner}@microsoft.com

**Abstract.** With the Distributed File System Replication component, DFS-R, as the central theme, we present selected protocol problems and validation methods encountered during design and development. DFS-R is currently deployed in various contexts; in Windows Server 2003-R2, Windows Live Messenger (Sharing Folders), and Windows Vista (Meeting spaces). The journey from an initial design sketch to a shipped product required mainly the dedicated effort of several testers, developers, program managers, and several others; but in some places cute problems related to distributed consensus and software model-checking emerged. This paper presents a few of these, including a distributed garbage collection problem, distributed consensus problems for reconciling tree-like data structures, using model-based test case generation, and the use of software model checking in design and development process.

## 1 Introduction

Designing and building distributed systems is challenging, especially if they need to scale, perform, satisfy customer functionality requirements, and, oh well, work. An example of a particularly challenging distributed system is multi-master, optimistic, file replication. One of the distinguished factors making distributed file replication hard is that file replication comes with a very substantial data component: the protocols need to be sufficiently aware of file system semantics, such as detecting and resolving name conflicting file creates and concurrent updates. Such races are just the tip of the iceberg. In comparison, cache coherence protocols that are known to be challenging to design, have a trivial data component, but to be fair have stronger consistency requirements.

Subtle protocol bugs can go (and have indeed gone) undetected for years due to the large number of interactions that are possible. With a sufficient number of deployments they *will* be encountered in the field, have costly consequences, and be extremely challenging to analyze. Our experience in developing DFS-R from the bottom up, is used to demonstrate several complementary uses of model-based techniques for system design and exploration. This paper provides

[1] To appear in Festschrift Symposium dedicated to the 70'th birthdays of Dines Bjørner and Zhou Chaochen, 24-25 September 2007, Macao, Editors Jim Woodcock and Cliff Jones

an experience report on these selected methods. Note that the material presented here reflect only a very partial view of the design and test of DFS-R.

DFS-R was developed to address correctness, scale, and management challenges encountered with a predecessor file replication product. Thus, the original impression was that we had the luxury of tackling a relatively well defined problem; to build a replication system specifically handling features of the file system NTFS, for replicating files between globally dispersed branch offices of corporations. Later on, it would turn out that DFS-R could be embedded within other scenarios, such as, in an instant messenger product. However, we consciously avoided over-loading with features from the onset. It means that DFS-R, for instance does not replicate files synchronously, only asynchronously (as it is meant for wide area networks); does not replicate general directed acyclic graphs, only tree-like structure; and does not maintain fine-grained tracking of operations, only state. While several such problems are interesting in other contexts, they did not fall into the scope of our original goals.

The organization of this paper follows the top-down design flow of DFS-R. The DFS-R system was originally conceived as a strictly state-based file replication protocol. Section 2 elaborates on the differences between state-based and operations-based replication systems. We developed a high-level state machine specification of DFS-R by using a transition system presented as a collection of guarded commands. The guarded commands were subsequently implemented as an applicative program in OCaml. This paved the way for performing efficient state space exploration on top of the design. Section 3 elaborates on the protocol, and Section 4 summarizes prototyping experiences. As the development took place, several assumptions made in the abstract design turned out to be unrealistic, and we redid the high-level design using the AsmL tools that were built at Microsoft for software modeling and test case generation. Section 5 elaborates on the experiences from using AsmL. A number of well-separated distributed protocol problems emerged during the development. Section 6 describes the distributed tree reconciliation problem, and how we used a model checker, Zing, to expose bugs in both protocol proposals and legacy implementations. Section 7 describes the distributed tombstone garbage collection problem and a solution to it. While one cannot expect to get anywhere without a high-level understanding of the protocols involved in DFS-R, it is equally unrealistic to expect developing a production quality system without addressing systems problems. We were thus faced with a potentially large gap between simplified protocol substrates and the production code. Encouraged by the ability of the model-based state space exploration to expose subtle interaction bugs we repeated the state space exploration experiment on top of the production core. The resulting backtracking search tool may best be characterized as a hybrid software model checking, run-time verification tool. It operates directly at the source code level. It uses techniques, such as partial order reduction to prune search and custom allocation routines to enable backtracking search. Section 8 describes the infrastructure we developed and the experiments covering $\frac{1}{2}$ trillion scenarios.

## 2 File Replication

The style of replication systems under which DFS-R falls into is surveyed extensively in [1]. We here summarize a few of the main concepts relevant for DFS-R. The problem that DFS-R solves is to maintain mirror copies of selected directories across large networks of servers. The directories that are selected for replication are called *replicated folders*. Files and directories within these directories may be created, modified, deleted, moved, or renamed at any of the mirror sites. It is the job of DFS-R to distribute changes, detect and reconcile conflicts automatically when they arise. Distributed replication systems can be categorized according to what problems they solve and how they solve them. Figure 1 summarizes some of the main design choices one has when designing a replication system.

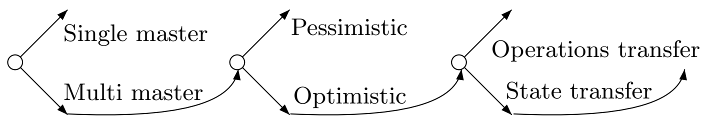

**Fig. 1.** Replication system ontologies

**Multi Master Replication** DFS-R is a multi-master replication system. Any machine may participate in changing resources, and their updates will have to be reconciled with updates from any other machine. A (selective) single-master system only replicates changes from a set of selected machines. All other machines are expected to maintain a mirror copy of the masters. This would mean that file system changes on non-masters would have to be reverted. If there is a designated master, one can even choose to maintain truth centrally. The challenge there is managing fail-over and network disconnects.

**Optimistic Replication** To support wide area networks (spanning the globe) DFS-R supports optimistic updates to files. This means that any machine may submit updates to resources without checking first whether the update is in conflict with other updates. Pessimistic replication schemes avoid concurrent update conflicts by serializing read and write operations using locking schemes.

**State and Operation transfer** A file system state is the result of the file operations (create, update, delete, move) that are performed on it. This suggests two approaches to realize file replication: intercept and replay the file operations, called operation transfer, or capture the file system state and replicate it as it is, called state transfer. DFS-R implements a state transfer protocol. There are several hard challenges with operations-transfer based systems. One is merging operations into a consistent serialization. Another, is space, as operations are not necessarily amenable to garbage collection.

**Perspective** There is no single choice of design parameters that handles all customer scenarios. In some configurations, corporations wish to designate machines as read-only, and can manage the additional constraints this leaves on network topologies. In other configurations there are reliable, but slow, wide-area networks and there is a need for file locking. With state transfer only, it is not possible to undo operations with arbitrary fine-grained control.

## 3 A State-based file replication system

In this Section we will outline the essence of a file replication system. While highly simplified, it reflects some of the early protocol design maneuvers.

### 3.1 Components

**Network** Abstractly, the problem at hand is to replicate files in a network of connected machines. Each machine maintains a file system view and a database. The network topology is indicated by a set of in-bound connections per machine. We assume a well-formed network, comprising of a digraph without self-loops, where each node is labeled by a unique machine identifier. A connected network is furthermore desirable for convergence.

$$\begin{array}{llll} nw & \in & \mathit{Network} & = \mathit{MachineId} \overset{m}{\mapsto} \mathit{Machine} \\ mch & \in & \mathit{Machine} & = \mathit{FileSystem} \times \mathit{DataBase} \times \mathit{inbound} \\ m & \in & \mathit{MachineId} & = \text{Globally unique identifier for a machine} \\ in & \in & \mathit{inbound} & = \mathit{MachineId}\textbf{-set} \end{array}$$

**File System** For our purposes, a file system is a collection of files each uniquely identified by an identifier, which is unique per file system. In NTFS, such identifiers are 64 bit numbers, called *file reference numbers*, on Unix-like file systems, these are called *inodes*. Each file has a file name, a parent directory and file data. One would typically expect a file system to be dictated as a tree-like structure comprising of files identified by file paths as strings, but this view turns out to be unsuitable for several reasons. For example, such a view is open to situations where files are moved around, such that the same path gets identified with completely different files. Identifying a file with an identifier makes it easier to support efficient replication of renaming directories with a large number of children, but makes it very hard to support merging contents from different directories. A file system is well-formed when the ancestral relations form a uniquely rooted connected tree (only the root has itself as a parent).

$$
\begin{array}{lll}
fs \;\in \mathit{FileSystem} & = & \mathit{FileId} \overset{m}{\mapsto} \mathit{FileRecord} \\
file \in \mathit{FileRecord} & = & \{name : \mathsf{Name},\ parent : \mathit{FileId}, data : \mathsf{Data}\} \\
fid \;\in \mathit{FileId} & = & \mathsf{Numeral}
\end{array}
$$

**Database** The file system only maintains information that is local to the machine where the files reside. In order to realize a file replication system, one needs to maintain information reflecting the shared state between machines. In DFS-R, this state is a database consisting of version vector and a set of records, one per replicated file.

$$
\begin{array}{lll}
(vv, rs) \in \mathit{DataBase} & = & \mathit{VersionVector} \times (\mathit{UID} \overset{m}{\mapsto} \mathit{IdRecord}) \\
vv \;\;\in \mathit{VersionVector} & = & \mathit{MachineId} \overset{m}{\mapsto} \mathsf{Numeral\text{-}set} \\
r \;\;\;\in \mathit{IdRecord} & = & \{fid : \mathit{FileId}, gvsn : \mathit{GVSN}, parent : \mathit{UID}, \\
 & & \quad clock : \mathsf{Numeral}, name : \mathsf{Name}, live : \mathsf{bool}\} \\
gvsn \in \mathit{GVSN} & = & \mathit{MachineId} \times \mathsf{Numeral} \quad \text{Global version sequence number} \\
uid \;\in \mathit{UID} & = & \text{Globally unique identifier}
\end{array}
$$

**Version vectors** File replication systems typically use global version sequence numbers, which are pairs (Unique Machine Identifier, Version Sequence Number), to identify a resource and its version globally. The version sequence number is a local time-stamp, which can be assumed monotonically increasing with changes. A version vector is a map from machine identifiers to version sequence numbers. They typically map a machine identifier to a single number, but in the case of DFS-R we found that allowing the vectors to map to a set of numbers (represented compactly as intervals of numerals) allowed handling, for instance, synchronization disruptions. Version vectors are also known as vector clocks. Version vectors are used to record a state of *knowledge*, as the vector indicates a water-mark of versions that have been received from other machines.

We may think of a version vector as a set of $GVSN$ pairs obtained by taking $\{(m, v) \mid [m \mapsto vs] \in vv \wedge v \in vs\}$. Similarly, one can form a version vector from a set of $GVSN$ pairs. In the future we will switch between the set and map view of version vectors depending on the context. Thus, $vv[m]$ is defined for each $m$. It is the empty set if $m \notin \mathrm{Dom}(vv)$ as a map.

**Database records** A file (we will use file to also refer to a directory) is identified globally through a unique identifier *uid*, while the per file system file identifier is the *fid*. The set of database records may be indexed by a *uid* and a *fid*. Each record stores a global version sequence number *gvsn* that tracks the version of the file, a file name *name*, a reference to a parent directory *parent*, and an indication whether the resource represents an existing file on the file system. If *live* is false, we call the resulting record a *tombstone*. The *clock* field is a Lamport clock [6], it gets incremented with every file update. Lamport clocks are used to enforce causal ordering per record by assuming a total lexicographic ordering on $GVSN$ and define:

$$r < r' \text{ iff } r.clock < r'.clock \vee (r.clock = r'.clock \wedge r.gvsn < r'.gvsn) \quad (1)$$

We will later establish that property (5), which only uses version vectors, suffices for detecting conflicts (absence of causality) among *all* replicated files. Nevertheless, this property is significant, as the number of replicated files in the context of DFS-R is much larger than the number of replicating machines.

The records in DFS-R contain a number of additional fields, such as file hashes, file creation time and file attributes.

**Local and Global Consistency** We are now in a position where we can state the main soundness properties that DFS-R aims to achieve:

– Global consistency: Databases of machines connected in a network are equal except for the contents of the *fid* fields.
– Local consistency: On each machine, the database records the content on the file system.

A very substantial part of DFS-R consists in maintaining local consistency. DFS-R uses the NTFS change journal, which for every file operation produces a record, accessible from a special file. The change journal presents an incremental way to obtain file changes. Since DFS-R only tracks files that are replicated, it furthermore needs to scan directories that are moved in and out of the replicated folders. Also, the local consistency algorithms need to take into account that change journals *wrap*, that is, not all consecutive changes are available for DFS-R, and that change journals are deleted, resized and/or re-created by administrators. We will here concentrate only on global consistency as it illustrates the distributed protocol problems later in this paper.

So for the rest of the discussion, we will use simplified definitions of machines and database records. While this approach makes things look much simpler than reality, it allows us to concentrate on the specific topics in this paper.

$$
\begin{array}{lll}
m \in \mathit{Machine} & = & \mathit{VersionVector} \times (\mathit{UID} \overset{m}{\mapsto} \mathit{IdRecord}) \times \mathit{inbound} \\
r \in \mathit{IdRecord} & = & \{\mathit{gvsn} : \mathit{GVSN}, \mathit{parent} : \mathit{UID}, \mathit{clock} : \mathsf{Numeral}, \\
 & & \quad \mathit{name} : \mathsf{Name}, \mathit{live} : \mathsf{bool}\}
\end{array}
$$

## 3.2 Operations

The main operations relevant to file replication consist of local file system activity and synchronization.

The file system operations called Create, Update, Rename and file Delete in Fig. 2. cause the local version vector to be updated with a fresh version for the machine that performs the change. The database records are also updated to reflect the new file system state.

We assume an initial state consisting of an arbitrary network of machines all sharing a single replicated root folder and no other files. We use tuples with mutable fields in the guarded commands, and we omit checks for whether elements are in the domain of a map prior to accesses.

A direct way to synchronize two data-bases is by merging version vectors and traversing all records on a sending machine $m_2$; those records whose keys

```
Create($nw, m, uid, parent, name$) :
    let $(vv, rs, in) = nw[m]$, $v = 1 + \max(vv[m])$
    assume $\forall [\_ \mapsto (\_, rs', \_)] \in nw \;.\; uid \notin rs'$ ($uid$ is fresh in $nw$)
      $rs[parent].live \;\wedge\;$ $name$ is fresh under $parent$
    $vv[m] := vv[m] \cup \{v\}$
    $rs[uid] := \{gvsn = (m, v), parent, name, clock = v, live = \texttt{true}\}$

Update($nw, m, uid$) :
    let $(vv, rs, in) = nw[m]$, $v = 1 + \max(vv[m])$, $clock = \max(v, rs[uid].clock + 1)$
    assume $rs[uid].live$
    $vv[m] := vv[m] \cup \{v\}$
    $rs[uid] := rs[uid]$ with $\{clock, gvsn = (m, v)\}$

Rename($nw, m, uid, parent', name'$) :
    let $(vv, rs, in) = nw[m]$, $v = 1 + \max(vv[m])$, $clock = \max(v, rs[uid].clock + 1)$
    assume $rs[uid].live \wedge rs[parent'].live \wedge$ $name'$ is fresh under $parent'$
       Rename maintains tree-shape of directory hierarchy
    $vv[m] := vv[m] \cup \{v\}$
    $rs[uid] := rs[uid]$ with $\{gvsn = (m, v), parent = parent', clock, name = name'\}$

Delete($nw, m, uid$) :
    let $(vv, rs, in) = nw[m]$, $v = 1 + \max(vv[m])$, $clock = \max(v, rs[uid].clock + 1)$
    assume $rs[uid].live \;\wedge\; (\forall uid' \in rs \;.\; rs[uid'].parent \neq uid \;\vee\; \neg rs[uid'].live)$
    $vv[m] := vv[m] \cup \{v\}$
    $rs[uid] := rs[uid]$ with $\{gvsn = (m, v), clock, live = \texttt{false}\}$
```

**Fig. 2.** Basic file system operations

do not exist on the receiving machine $m_1$ are inserted. Records, that dominate existing records on $m_1$ are also inserted. Fig. 3. illustrates the proposed scheme. The scheme implements a *last-writer wins* strategy, as later updates prevail over earlier updates. We will later realize that the check $v \notin vv_1[m]$ is in fact redundant. Another property of this scheme is that each update is processed independently. Notice that this is an implementation choice, which comes with limitations. Conflict resolution that can only perform decisions based on a single record cannot detect that a machine swapped the names of two files. Namely, suppose machine $m_1$ and $m_2$ share two files named $a$ and $b$. Then $m_2$ renames $a$ to $c$, $b$ to $a$, then $c$ to $b$. The names of the two files are swapped, but each record is name conflicting with the configuration on $m_1$. So when $m_1$ synchronizes with $m_2$, it will be resolving two name conflicts instead of performing the swap.

Our first observation is that the resulting system maintains a basic invariant: the versions of all records are tracked in the version vectors.

$$\forall [\_ \mapsto (vv, rs, \_)] \in nw,\; [\_ \mapsto \{gvsn = (m, v)\}] \in rs \;.\; v \in vv[m] \qquad (2)$$

Thus, a more network efficient version of BasicSyncJoin proceeds by

1. The receiving machine $m_1$ gets version vector $vv_2$ from the sender $m_2$.

```
BasicSyncJoin($nw, m_1, m_2$)
    let $(vv_1, rs_1, in_1) = nw[m_1]$
    let $(vv_2, rs_2, in_2) = nw[m_2]$
    assume $m_2 \in in_1$
    for each $[uid \mapsto r] \in rs_2$:
        let $(m, v) = r.gvsn$
        if $v \notin vv_1[m] \ \wedge \ (uid \notin rs_1 \ \vee \ rs_1[uid] < r)$ then
            $rs_1[uid] := r$
    $vv_1 := vv_1 \cup vv_2$
```

**Fig. 3.** Simplified synchronization

2. It then subtracts $vv_1$ from $vv_2$, forming $vv_\Delta := vv_2 \setminus vv_1$.
3. The sending machine is asked for records whose versions are in $vv_\Delta$.
4. The database of $m_1$ is updated as in BasicSyncJoin.

The more refined version of global consistency we seek can also be formulated in terms of version vectors, namely, that databases coincide on all shared versions:

$$r_1.gvsn \notin vv_2 \ \vee \ rs_2[u] = r_1 \ \vee \ rs_2[u].gvsn \notin vv_1, \tag{3}$$

for every $m_1, m_2$, such that $(vv_1, rs_1, \_) = nw[m_1]$, $(vv_2, rs_2, \_) = nw[m_2]$ and $[u \mapsto r_1] \in rs_1$.

So far our transitions ensure that all versions in the version vectors are consecutive,

$$\forall [m \mapsto vs] \in vv \ .vs = \{1, \ldots, \max(vs)\}. \tag{4}$$

The second observation is a basic property of the system: concurrent updates to the same resource may be detected by at least one machine during BasicSyncJoin. Suppose that $m_1$ and $m_2$, and $uid$ are such that $(vv_1, rs_1, \_) = nw[m_1]$, $(vv_2, rs_2, \_) = nw[m_2]$, and $[uid \mapsto r_1] \in rs_1$, $[uid \mapsto r_2] \in rs_2$. When $m_1$ installs $r_2$ we would like to know whether $r_2$ was derived from $r_1$, or if $r_2$ was obtained concurrently with $r_1$. The answer to whether $r_2$ is concurrent with $r_1$ turns out to be simple; $r_2$ is concurrent with $r_1$ iff the version of $r_1$ is not known to $m_2$:

$$r_1.gvsn \notin vv_2 \tag{5}$$

To prove this property, we can add a history variable $rs_{all}$ to each machine. The history variable $rs_{all}$ is a set of all records ever maintained by the machine. If one prefers, one may view this as specifying the cone of causality. Every update to the main set of records $rs$ gets reflected by adding the updated record to $rs_{all}$. In the operation BasicSyncJoin, take the union of $rs^1_{all}$ and $rs^2_{all}$. Now observe that invariant (2) also holds for $rs_{all}$.

Detection of concurrent update conflicts is useful when one wants to perform conflict detection and resolution, either manually or automatically. DFS-R performs the conflict resolution automatically, as replication is a continuous service, but stores conflicting files in a designated folder. Conflict resolution is performed on version vectors, so once a machine has performed conflict resolution and merged version vectors, the conflict is no longer visible to other machines. A different scheme that works for detecting conflicts is by associating a hash-tree [7, 8] comprising of hashes of all the previous versions of a file. The size of the unrolled hash-tree is then proportional to the number of changes to the file, while version vectors grow proportionally to the number of machines. If machines are reasonably well synchronized, they do not need to unroll large hash trees from remote peers.

### 3.3 The Real Deal with Join

The use of BasicSyncJoin is insufficient for file replication. There are two fundamental flaws and limitations: First, it allows installing updates that conflict with file system semantics: it may introduce orphaned files without parent directories, mark non-empty directories as tombstones, create multiple files in the same directory with the same name, and introduce cyclic directory structures. Second, BasicSyncJoin processes all updates in one atomic step. This is unrealistic in the presence of network outages and continuous file system activity. DFS-R realizes non-atomic joins by committing only versions from the processed records on disconnects (instead of all of $vv_2$). It also pipe-lines multiple joins should the sending machine create new updates while (large) files from previous updates are still being downloaded. A consequence of this relaxation is that condition (5) is only a necessary, but not sufficient condition for conflict detection. Invariant (4) does not hold either, but this is insignificant, as we introduced sets in the range of version vectors to deal with partial synchronization. Fig.4. illustrates the additional refinements one needs to add to BasicSyncJoin in order to address file system semantics. We have limited the iteration of database records to $vv_2 \setminus vv_1$ to reflect invariant (2), which still holds. We abstain from illustrating the non-atomic, pipe-lined version.

The refined SyncJoin mentions auxiliary functions *conflict-winner*, *purge-losers*, and *revert-update*. The definition and analysis of these is the subject of Section 5, but here, we will summarize some of their requirements.

**Non-interference** It is trivial to realize a convergent, consistent file replication system that just deletes all files. So, obviously, we would like to ensure that DFS-R does not touch the file system on its own. Requiring that DFS-R not delete or move around any files is too restrictive, because a system that must automatically resolve conflicts will have to handle creation of name conflicting files and directories.

**Re-animation** A basic (user) requirement for DFS-R was that directories cannot be deleted if they contain files that have not been processed by the deleting party. Thus, re-animation requires that files and even directories in a transitive

```
SyncJoin$(nw, m_1, m_2)$
  let $(vv_1, rs_1, in_1) = nw[m_1]$
  let $(vv_2, rs_2, in_2) = nw[m_2]$
  assume $m_2 \in in_1$
  for each $[uid \mapsto r] \in rs_2$ where $r.gvsn \in vv_2 \setminus vv_1$:
    if $uid \notin rs_1 \ \vee \ rs_1[uid] < r$ then
      if $\mathit{conflict\text{-}winner}(m_1, r)$ then
        $\mathit{purge\text{-}losers}(m_1, r)$
        $rs_1[uid] := r$
      else $\mathit{revert\text{-}update}(m_1, r)$
  $vv_1 := vv_1 \cup vv_2$
```

**Fig. 4.** Synchronized join

way get re-created. They are re-created to preserve content that was created or modified independently of the (remote) deletion.

**Convergence** A key property of replication is obviously that all replica members should converge to the same mirror image when there are no independent updates to the replica sets. In general one cannot check convergence as an invariant. However, as our experience with Zing (Section 6) illustrates, it is possible to find divergence bugs by checking invariants that imply convergence.

**Feature interaction** One of the hard problems with designing a distributed application, like DFS-R, is taking feature interaction into account. Features that are not directly related may interact in unpleasant ways when composed. An illustrative example of two features that interact comprises of re-animation and name-conflict resolution. Name conflict resolution has in DFS-R the side effect of soft deletion. The name conflict loser gets moved to a conflict area, but from the point of view of replication it is deleted. These two features do not compose: a directory may lose a name conflict and be deleted, but a modification to a child file may require the name conflicting directory to be re-animated. Consequently, DFS-R has to take such conflicts into account.

## 4 Prototyping DFS-R with OCaml

Section 3 illustrated a simple file system model and replication protocol. As features and requirements were added, the complexity of the problem rose, and we could see that invariants were easily broken. We therefore found that an informal design would be insufficient in convincing us and our peers to the soundness of any protocol proposal, so we developed a prototype system in OCaml.

Of particular interest was that the OCaml prototype supported both a *simulation* and a *realistic* mode. In simulation mode, the replication system would manipulate in-memory data structures for file systems, and data-bases. In realistic mode, the prototype accessed files on NTFS and updated a persistent store.

Neither mode was using a network, so all remote procedure calls would be performed by local procedure calls, and multiple copies of DFS-R ran in the same process. The simulation mode was furthermore applicative in all essential operations. This made implementing a backtracking search over the actions trivial. Operations in simulation mode were several orders of magnitude faster. We also added ad-hoc partial order reduction techniques, and performed massive simulations on top of the synchronization core. Prior to starting the implementation of DFS-R we thus covered some 120 billion scenarios each comprising of 16 file and synchronization actions. Section 8 elaborates on similar methods used for the production core.

## 5 Modeling DFS-R with AsmL

The OCaml prototype soon diverged from the implementation, as constraints, such as database layout changed. It was also inadequate for documenting the protocol at an abstract, yet sufficiently faithful level. We therefore turned to AsmL [9], developed at MSR, for describing the DFS-R protocol. The very readable format of AsmL and the integration with Microsoft Word was particularly useful in our context, as we aimed at a specification which could be read by newcomers to the DFS-R project. Today the AsmL specification serves as the main high-level, executable, overview of DFS-R. We will not repeat the detailed AsmL specification here, as it takes around 100 pages. To give the flavor, Fig.5. summarizes the data types available per replicating machine.

The AsmL description follows the componentization and protocol design in a top-down fashion. At the top-level, the design describes the main modules that comprise synchronizing machines. For the case of DFS-R, this is encapsulated by a Machine class, which contains the main components.

```
class Machine
   machineId as MachineId // Unique identifier to distinguish machine
   var fs as FileSystem     // File system interface
   var db as Database       // Persistent database that DFS-R maintains
   var uc as UsnConsumer // Consuming USN records from the NTFS journal
   var dw as DirWalker      // Walking directories to update the database
   var inbound as Map of MachineId to InConnection
   var outbound as Map of MachineId to OutConnection

class InConnection           // State relevant for an incoming connection
class OutConnection          // State relevant for an outgoing connection
```

**Fig. 5.** Replicating machine in AsmL

### 5.1 Protocol description

The AsmL specification elaborates further on fleshing out the contents of the machine components. The main reactive components that are modeled in detail are the (1) consumption of file system events and their effect on the local database, and (2) the main synchronization handshakes.

### 5.2 Test case generation

The resulting model is sufficiently detailed to describe the behavior of DFS-R based on local file system events as well as distributed synchronization events. This allows defining virtual networks of machines that can be composed and simulated within AsmL. In particular, we hooked up the FSM generation tool of AsmL and generated test sequences. Lacking tight .NET integration with DFS-R, we resorted to using the FSM generation tool generate test cases in an XML file and implement a reader that interprets the traces within DFS-R.

## 6 Reconciling Trees

In this Section we illustrate the use of a model-checker Zing [10] for checking conflict resolution strategies for concurrent moves. Recall that one of the requirements for DFS-R was that it replicate and maintain directory hierarchies as tree-like structures. When machines are allowed to move files around on a network, it may however be possible arriving at configurations that cannot be reconciled into a directory tree. Fig.6. illustrates an instance of this problem: two machines share directories $a$, $b$, and $c$. One machine creates the tree $a \to b \to c$, the other $c \to b \to a$. What should $b$'s parent be?

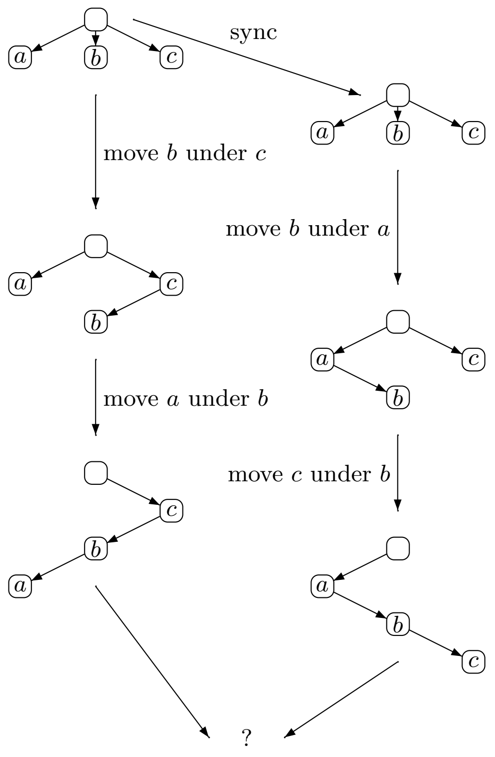

**Fig. 6.** Concurrent conflicting moves

We used Zing to check for convergence of a proposed resolution method for concurrent moves. Zing demonstrated that the proposal was open to divergence by producing a counter-example along the lines of Fig.6. The counter-example found by Zing, was subsequently tested against another shipped replication product which failed to converge. This bug had gone undetected for several years.

## 6.1 Zing

Zing is a model checker for multi-process programs. Zing's input language is perhaps easiest compared with Promela [11]. The notion of process and atomic statements are similar, while Zing appeals to an object oriented programming style.

## 6.2 A Zing encoding

Our Zing encoding of the tree reconciliation problem uses the absolute minimal features necessary to emulate conflict resolution of concurrent moves. Thus, each machine maintains $n(= 3)$ resources, each resource is identified by a number $0, \ldots, n-1$, and a designated root folder is identified by the number $-1$. A resource has a *parent*, which is a number $-1, 0, \ldots, n-1$, and a *clock* used to impose a total ordering on resources. Resources can be updated by changing the *parent* and *clock*, but only if the update does not introduce a self-loop. Fig.7. contains the minimal amount of Zing declarations to define two machines each with three resources all residing under a common root.

A model checker is well suited at specifying a set of possible configurations implicitly by using non-deterministic choices. Thus, we arrive at a non-flat configuration by first moving files around randomly on each machine, with each move incrementing globalClock and using its value as the clock on the moved resources.

It remains to define synchronization in Zing. Our model for synchronization is that machines send the state of a random node to a random machine. It requires the recipient to determine the outcome based on the state of a single node. Thus, traces can be identified as sequences of triples

$$\langle \mathsf{node}_1, \mathsf{src}_1, \mathsf{dst}_1 \rangle, \langle \mathsf{node}_2, \mathsf{src}_2, \mathsf{dst}_2 \rangle, \ldots,$$

where node is the index of a node, the content of the node on the source machine is given by src, and dst is a machine that should reconcile the node. The synchronization protocol will need to implement a function, sync, which based on a triple, updates the state of dst. The problem is furthermore narrowed down as we prescribe sync should use the clock numbers to implement a last writer wins-by-default strategy. Unfortunately, the last writer cannot win unconditionally if the update introduces a cycle, and the remaining problem is to find a routine resolve, which applies the update, but does not introduce a cycle. We can check whether an implementation of sync converges by setting a bound on globalClock and systematically examining each possible trace.

In the following we will examine a few proposals for resolve. We examined several others, but the ones given here are sufficiently illustrative.

**Priority inversion** is a tempting solution. The clock on the destination machine is increased to dominate the clock of the source node.

```
class Node {
    int parent;
    int clock;
};
array Tree[3] Node;

class Machine {
    Tree tree = new Tree{{-1,0},{-1,0},{-1,0}};

    atomic bool cycle(int node, int parent) {
        return (parent != -1) &&
            (parent == node || cycle(node, tree[parent].parent));
    }

    atomic void move(int node, int parent, int clock) {
        assume(!cycle(node, parent));
        tree[node].parent = parent;
        tree[node].clock = clock;
    }
};
array Machines[2] Machine;
static Machines machines;
static int globalClock = 0;
```

**Fig. 7.** Replicating machine in Zing

```
static atomic void resolve(int node, Node src, Machine dst) {
    dst.tree[node].version = ++globalClock;
}
```

Not only is it not obvious whether this solution is correct, but it is also wrong. Zing found a two-machine counter-example by searching 1.5 million states in 4-5 minutes (on a 2GHz, 512MB Dell Optiplex). The counter example essentially consisted of the configuration from Fig.6. Divergence is exercised when the two machines ping-pong the directory *b* to each other.

**Intentional grounding** moves conflicting nodes to the root.

```
static atomic void resolve(int node, Node src, Machine dst) {
    dst.move(node, -1, ++globalClock);
}
```

This solution works (and works for Zing too), but it is overly pessimistic, as it may move directories from deeply nested positions directly to the root. Within the context of file systems, where directories have controlled access (using access control lists, ACLs) this furthermore imposes security problems.

```
class Sync {
    static atomic void sync(int node, Node src, Machine dst) {
        if (src.clock > dst.tree[node].clock) {
            if (dst.cycle(node, src.parent)) {
                Sync.resolve(node, src, dst);
            } else {
                dst.move(node, src.parent, src.clock);
            }
        }
    }

    static void synchronize() {
        while (!Sync.allInSync()) {
            assert(globalClock <= maxClock);
            int src, dst = choose(0..1);
            int node = choose(0..2);
            assume(src != dst);
            Sync.sync(node, machines[src].tree[node], machines[dst]);
        }
    }

    atomic bool allInSync(); // true if the trees of all machines are equal
```

**Fig. 8.** Synchronization core in Zing

**Permutation** does not move conflicting nodes directly to the root, but moves them beneath the immediate parent.

```
static atomic void resolve(int node, Node src, Machine dst) {
    if (dst.tree[node].parent != -1) {
        dst.move(node, dst.tree[dst.tree[node].parent].parent, ++globalClock);
    }
    else {
        dst.tree[node].version = ++globalClock;
    }
}
```

While less pessimistic, it also suffers from security problems with access control: the scheme allows moving directories to places they have never been moved by any replicating machine.

**Parental demotion** Another appealing approach is to accept the instruction as is, but if the instruction introduces a directory cycle, then move the new parent under the previous parent of the node.

```
static atomic void resolve(int node, Node src, Machine dst) {
    dst.move(src.parent, dst.tree[node].parent, ++globalClock);
    dst.move(node, src.parent, src.clock);
}
```

Unfortunately, we were able to find a configuration where this scheme diverges. The smallest example we were able to find consists of 6 machines each with 3 directories. It requires a careful coordination between the machines to exercise divergence. This time we had to find the counter-example manually. The state space in this case proved larger than what Zing could handle.

Suppose initially:
$m_1 : a \to b \to c$, clocks $= \{a \mapsto 0, b \mapsto 1, c \mapsto 2\}$. That is, $m_1$ has $a$ under the replicated folder root, $b$ under $a$, and $c$ under $b$. The clock of $b$ is set to 1, and $c$'s clock is set to 2.
$m_2 : b \to c \to a$, clocks $= \{b \mapsto 0, c \mapsto 11, a \mapsto 3\}$.
$m_3 : c \to a \to b$, clocks $= \{c \mapsto 0, a \mapsto 4, b \mapsto 5\}$.
$m_4 : a \to c \to b$, clocks $= \{a \mapsto 0, c \mapsto 6, b \mapsto 7\}$.
$m_5 : c \to b \to a$, clocks $= \{c \mapsto 0, b \mapsto 12, a \mapsto 8\}$.
$m_6 : b \to a \to c$, clocks $= \{b \mapsto 0, a \mapsto 9, c \mapsto 10\}$.

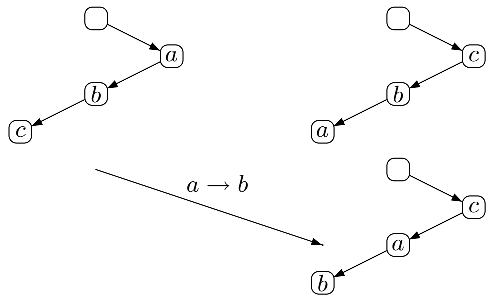

**Fig. 9.** Parental demotion.

$m_5$ sends $b$ to $m_1$, $m_1' : a \to c \to b$, clocks $= \{a \mapsto 0, b \mapsto 12, c \mapsto 13\}$
$m_2$ sends $c$ to $m_4$: $m_4' : a \to b \to c$, clocks $= \{a \mapsto 0, b \mapsto 14, c \mapsto 11\}$
$m_1'$ sends $c$ to $m_2$: $m_2' : b \to a \to c$, clocks $= \{b \mapsto 0, c \mapsto 13, a \mapsto 15\}$
$m_4'$ sends $b$ to $m_5$: $m_5' : c \to a \to b$, clocks $= \{c \mapsto 0, a \mapsto 16, b \mapsto 14\}$
$m_2'$ sends $a$ to $m_3$: $m_3' : c \to b \to a$, clocks $= \{c \mapsto 0, b \mapsto 17, a \mapsto 15\}$
$m_5'$ sends $a$ to $m_6$: $m_6' : b \to c \to a$, clocks $= \{b \mapsto 0, c \mapsto 18, a \mapsto 16\}$

That state is isomorphic to the starting state using the correspondence:

$$\{m_1 \mapsto m_4', m_2 \mapsto m_6', m_3 \mapsto m_5', m_4 \mapsto m_1', m_5 \mapsto m_3', m_6 \mapsto m_2'\} \quad (6)$$

At this point, $m_6'$ can take the role of $m_2$, and $m_3'$ can take the role of $m_5$ to kick off another round.

## 6.3 A mountain too high for Zing?

An attempt was made to extract a more realistic Zing model of DFS-R by using the AsmL specification, and perform comprehensive model checking of the full synchronization core. The resulting 3400 line model was then exercised by Zing, which checked the model for consistency. Unfortunately, the resulting state space was vastly larger than what Zing could reasonably handle. The most effective way we know of performing state space exploration of DFS-R therefore remains the depth-bounded search presented in Section 8.

## 6.4 The Zing Experience

This Section illustrated the concurrent directory move problem in the context of using a state-space exploration tool, Zing, for checking design proposals. The concurrent move problem is interesting in its own right, but the main take-away here is that state space exploration tools, such as Zing, are valuable for experimenting with design ideas on protocol substrates. Our take-away was to remain using priority inversion as the conflict resolution mechanism in DFS-R. To avoid divergence we imposed stronger restrictions on the order of processing updates.

## 7 Distributed Garbage collection

Our presentation of DFS-R has so far no mechanism to garbage collect database records for deleted files. We need the database records so that file deletion can be replicated in a timely manner, but when all replicating machines agree that a file has been deleted, it should in principle be possible to remove the tombstone. Prior solutions to detecting when to delete dead resources involve two way commit protocols [12–14] to either agree on the when to add machines to a network, or when to safely collect resources marked for deletion. Solutions in replication systems, such as, Clearinghouse [15], NTFRS and other replications systems use a timeout based collection of tombstones: If a record has been marked as tombstone for 30 or 60 days, simply delete it from the database. Fig. 10 contains the corresponding transition that performs the garbage collection non-deterministically.

```
GarbageCollect$(nw, m, u)$ :
    let $(vv, rs, in) = nw[m], \quad r = rs[u]$
    assume $\neg r.live$
    $rs := rs \setminus [u \mapsto r]$
```

**Fig. 10.** Tombstone garbage collection

This solution does not address deleting content on machines that have been disconnected beyond the timeout value of the tombstones. While this situation reflects lack of consensus it also widens the likelihood that this content may reappear in other machines if the offline machine later makes changes to this content or if the machine has to recover from database loss.

It turns out that with synchronous joins we have a necessary and sufficient basis for detecting tombstones. The key is, as in (2), that version vectors maintain a trace of previously observed changes.

Let $m_1, m_2$ be machines, such that $(vv_1, rs_1, \_) = nw[m_1]$, $(vv_2, rs_2, \_) = nw[m_2]$ and $[u \mapsto r_1] \in rs_1$. The resource $r_1$ is subsumed by a tombstone if we have:

$$r_1.gvsn \in vv_2 \ \wedge \ \left[ u \notin rs_2 \ \vee \ \begin{pmatrix} \wedge \ rs_2[u].gvsn \in vv_1 \\ rs_2[u].gvsn \neq r_1.gvsn \end{pmatrix} \right] \tag{7}$$

The system with SyncJoin, GarbageCollect and the file system operations satisfies the following invariant: Whenever (7) holds, then some time in the past, there is a machine $m_3$, with a tombstone for $u$ that dominates the resource $r_1$; or with notation, if (7), then previously

$$\exists m_3, rs_3 \ . \ (\_, rs_3, \_) = nw[m_3] \ \wedge \ \neg rs_3[u].live \ \wedge \ rs_3[u] \geq r_1 \tag{8}$$

Conversely, if (7) is false, then for $[u \mapsto r_1]$ either $m_2$ does not know about $r_1$, or $m_2$ has a resource that $m_1$ does not know about. Regular synchronization takes care of reconciling the state in these cases.

This property suggests a secondary protocol for asynchronous garbage collection: periodically retrieve the version vector and all records from a partner machine, then garbage collect live records whenever condition (7) holds. DFS-R implements such a secondary protocol, but we observed that condition (7) in the presence of non-atomic joins is no longer necessary and sufficient for detecting missed tombstones. In general, condition (7) is only sufficient for detecting when the standard join does not ensure convergence.

## 8 Implementation checking

A common theme in the previous sections has been that we could take advantage of somewhat subtle properties of a simple transition system to achieve goals, such as garbage collection, conflict resolution, and reconciling concurrent rename conflicts; but small modifications could break everything, and innocent looking solutions could be broken in complicated ways.

In view of the complexity of the problem and the encouraging results with the OCaml prototype we therefore decided to simulate the production version of the synchronization core of DFS-R using model-checking techniques. This Section describes the components that comprise the simulator. In summary, the simulator works by exercising different combinations of file system operations followed by synchronization steps in alternation, then it backtracks to visit different combinations. Some traces get pruned by partial order reduction techniques. This drastically reduces redundancies in the search tree. Backtracking search requires replacing the memory layer such that old state can easily be retrieved. To relieve the simulator from suspending threads at arbitrary points we ensure that the core makes use of suitable thread abstraction allowing it to single step through large non-blocking atomic units. Finally, certain components are abstracted to gain speed and control.

Thus, the main ingredients in our software model-checking experiment were:

1. Identifying a suitable vocabulary of actions to exercise.

2. Providing a memory layer that supports efficient backtracking.
3. Providing a threading layer that supports context switching and control of which threads run.
4. Virtualize components that change device state.
5. Prune search using partial order reduction techniques.

Ideally, one would like a general framework to be able to handle simulating systems, such as DFS-R. At the time we developed the framework, nothing suitable was available. Since then, efforts have been made to address problems like ours [16] using general frameworks.

## 8.1 Vocabulary

The simulation layer executes tasks in all possible inter-leavings. We will describe tasks in more detail below, as they are used as a thread layer. A task step defines an atomic action. Simulating DFS-R requires providing a handle into the atomic actions that a machine may perform, but of equal importance also provide environment actions, such as file system operations. The actions that are presented to the simulator are summarized as $\mathcal{L}_{action}$. Finally, we can define a simulation trace $\mathcal{S}_{trace}$ as a sequence $d$ of actions. In our experiments we set $d = 16$, with the assumption that most bugs could be exercised with a few number of operations.

$$
\begin{array}{lcll}
\Sigma_{file} & = & \{a, b, c, d\} & \text{vocabulary of files} \\
\Sigma_{dir} & = & \{p, q, r\} & \text{directories} \\
\Sigma_{res} & = & \Sigma_{file} \cup \Sigma_{dir} & \text{resources} \\
\mathcal{L}_{fs} & = & \{\mathit{share}/, \mathit{noshare}/\}(\Sigma_{dir}/)^{*}\Sigma_{res} & \text{file paths} \\
\Sigma_{m} & = & \{m_1, m_2, m_3, m_4\} & \text{machines} \\
\mathcal{L}_{action} & = & \mathsf{rename}(\Sigma_m, \mathcal{L}_{fs}, \mathcal{L}_{fs}) & \text{actions} \\
 & \cup & \mathsf{create}(\Sigma_m, \mathcal{L}_{fs}) & \\
 & \cup & \mathsf{update}(\Sigma_m, \mathcal{L}_{fs}) & \\
 & \cup & \mathsf{delete}(\Sigma_m, \mathcal{L}_{fs}) & \\
 & \cup & \mathsf{sync}(\Sigma_m, \Sigma_m) & \\
 & \cup & \mathsf{read{-}journal}(\Sigma_m, N \cup \{\omega\}) & \\
\mathcal{S}_{trace} & = & \mathcal{L}_{action}^{d} & \text{simulation trace}
\end{array}
$$

The set of possible file system operations are generated using a finite alphabet of file and directory names. They may take place on one of the machines listed in $\Sigma_m$. The internal actions of DFS-R are split into two sets: (1) reading the USN journal for 1, 2, 3, etc. steps or until reaching a fix-point ($\omega$ steps); (2) synchronizing symmetrically between two machines. Paths starting with *share* are replicated, paths starting with *noshare* are outside the replicated folder.

## 8.2 A custom memory layer

Key to supporting efficient backtracking is to be able to save and restore state. Our simulation does not backtrack over suspensions within stack frames. This limitation allowed us to concentrate on tracking heap allocated memory only. In summary, the simulation environment saves aside a copy of the heap when entering a new backtracking point for the first time. When re-entering the backtracking point, it can dispense all memory allocated within the branch and restore the previous state. Unfortunately, not all heap-allocated memory can be reclaimed at backtracking points. In particular, memory that is associated with device state cannot just be overwritten on backtracking points. For instance, buffers that are allocated by procedures that print to files cannot be reclaimed using a stack discipline. This led to a dual mode custom memory layer, one for backtracking mode and one for non-backtracking mode.

## 8.3 Thread layering

It is challenging in itself to model check multi-threaded programs faithfully. A real software model checker would have to allow context switches at arbitrary control locations. The first problem requires infrastructure; the model checker will have to save and restore the stack. This amounts to mirroring thread context switches. A more fundamental problem is the significant increase in the state space as every program counter is potentially a backtracking point.

We bypassed these issues by implementing a thread abstraction that wrapped around thread pools and timer queues. Both facilities are supported by operating system APIs, but require the caller to maintain different context depending on whether a job is spawned directly in a thread pool or delayed in a timer queue.

Our thread layer combines these two concepts into a single *task* entity, which can be set to run immediately, or with a non-zero delay. To support simulation, tasks support dual modes: one for running in a multi-threaded environment, and another for running in a single-threaded simulation environment.

## 8.4 Virtualization

The interfaces to the on-disk database, the file system, and the network layer had to be abstracted and re-coded for speed and control. The abstractions were indispensable in making simulation practical. Creation time of a fresh on-disk JET-blue database takes for instance 2 seconds (it creates several larger files, including logs). Backtracking over such disk operations would slow down simulation to a crawl.

Using abstractions also came with several limitations. Foremost, bugs inside the physical modules were not exposed by simulation, as they were simply not exercised. It was also limited what we found worth to reflect in an abstraction. The database abstraction did thus not implement the ACID properties. This was reasonable as all transactions in DFS-R are short lived, but this prevented us identifying code paths that would lead to conflicting updates. Such errors were only later exposed during stress runs.

## 8.5 Partial Order Reduction

The set of simulation traces introduced in Section 8.1 contain a large number of essentially symmetric traces. For example, the order of creating files *share/p/a* and *share/p/b* on the same machine is insignificant.

More precisely, let $\pi, \pi' \in \mathcal{L}_{fs}$ be file paths, then we define $\pi \perp \pi'$ (read as $\pi$ is orthogonal to $\pi'$) as a binary relation on paths if neither path is a prefix of the other. Furthermore, let $m$, $m'$ be machines, $op_1$, $op_2 \in \{\mathsf{create}, \mathsf{delete}, \mathsf{update}\}$, then we define the orthogonality relation $\perp$ on actions by:

$$op_1(m, \pi) \perp op_2(m, \pi') \text{ if } \pi \perp \pi'$$

In general, actions are considered orthogonal if they reside on different machines, thus:

$$op_1(m, \pi) \perp op_2(m', \pi') \text{ if } m \neq m' \ \wedge \ op_1, op_2 \in \{\mathsf{create}, \mathsf{delete}, \mathsf{update}, \mathsf{rename}\}$$

We overload the use of $\perp$ to also capture idem-potency of actions. Actions that can be considered idempotent, such as two consecutive updates to the same file, are added to the relation.

Partial order reduction, based on $\perp$, is implemented by representing the vocabulary of actions $\mathcal{L}_{action} = \{a_1, \ldots, a_m\}$ using an mapping *por* from $\{1, \ldots, m\}$ into $2^{\{1,\ldots m\}}$ such that $a_i \perp a_j$, if and only if $j \in por(i)$. The sets $por(i)$ are implemented as bit-vectors, as $m$ is relatively small and of fixed size. Depth first search then prunes action sequences containing the pair $a_i a_j$ if $a_i \perp a_j$ (that would be $j \in por(i)$) and $j \leq i$.

## 8.6 Experiments

We ran simulation relatively early in the development process. As the product got more stable we ran a two week experiment, distributing the search over a cluster of 200 machines each exploring a different portion of the search space. This helped us covering slightly more than $\frac{1}{2}$ trillion scenarios, for checking main consistency properties.

Early on, simulation caught a large number of bugs that may have been caught later in stress. On the other hand, simulation served as a pretty good regression test as the implementation was evolving at a rapid pace. Some of the bugs found during simulation had the traits of being extremely difficult for a stress test to identify. For instance, the trace below exposed a corner case in the interplay between the components that recycled unique identifiers and those that walked directories.

$\mathsf{create}(m_2, noshare/p)$
$\mathsf{rename}(m_2, noshare/p, share/p)$
$\mathsf{read{-}journal}(m_2, \omega)$
$\mathsf{sync}(m_1, m_2)$
$\mathsf{rename}(m_2, share/p, noshare/p)$

$\mathsf{rename}(m_2, \mathit{noshare}/p, \mathit{share}/p)$
$\mathsf{read{-}journal}(m_2, \omega)$
$\mathsf{update}(m_1, \mathit{share}/p)$
$\mathsf{sync}(m_1, m_2)$

In comparison, the bugs found in stress were predominantly in the components that are abstracted away during simulation. There were a couple of exceptions, though. Stress exposed a divergence bug that simulation was blind to, it also exposed a protocol error exposed by asynchronous message passing.

## 9 Conclusions

This paper provided an experience report on the design and modeling used for the development of DFS-R. We put emphasis on the use of research tools Zing and AsmL from MSR, taking advantage of applicative features in garbage collected functional languages, and experiences with software model checking. None of these approaches are mainstream in product development, neither can it be said that DFS-R is a mainstream product; but we feel that pieces and variants of the approaches taken for DFS-R are benefitial for developing other distributed systems. The AsmL (now called SpecExplorer) and Zing tools are directly available as general purpose tools. Today, SpecExplorer is mainly directed towards model-based testing, while its use as a design tool is under-emphasized. Our software model checker was built exclusively for DFS-R. I doubt much of our simulation implementation can or should be re-used, as a custom simulation layer is easiest developed by the component owner. There is a ray of hope in future availability of general purpose tools for software model checking concurrent and/or distributed systems, though the task of building these is huge.

It should be noted that modeling, model exploration and software model checking are resource-wise minor activities in the larger picture of developing a product. Far more prolific was stress testing, where multiple instances of DFS-R are run against random file system operations. During development, each developer and tester ran stress sessions with up to 1-2 million file system operations every night. Besides stress runs, BVT regression tests, interoperability testing and bug-bashes each contributed in driving quality.

**Acknowledgements** DFS-R is the result of several people's hard and diligent work. I am indebted to having had the pleasure to work with Shobana Balakrishnan, Dan Boldo, Richard Chinn, Brian Collins, Scott Colville, Stone Cong, Huseyin Dursun, Ben Fathi, David Golds, Huisheng Liu, Ram Natarajan, Joe Porkka, Robert Post, Rafik Robeal, Christophe Robert, Masood Siddiqi, Guhan Suriyanarayanan, Dan Teodosiu; and several others from the Core File Systems group, Windows Live, Windows Networking and MSR. Thanks to Neil Conway for implementing the memory optimizations in the backtracking search, to Rostik Yavorsky for developing large parts of the AsmL specification, proving the garbage collection conditions correct, to Abhay Gupta for taking the AsmL specification and running it through Zing.

## References

1. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. **37**(1) (2005) 42–81
2. Pierce, B.C., Vouillon, J.: Unison: A file synchronizer and its specification. In Kobayashi, N., Pierce, B.C., eds.: TACS. Volume 2215 of Lecture Notes in Computer Science., Springer (2001) 560
3. Petersen, K., Spreitzer, M., Terry, D.B., Theimer, M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: SOSP. (1997) 288–301
4. Malkhi, D., Terry, D.B.: Concise version vectors in winfs. In Fraigniaud, P., ed.: DISC. Volume 3724 of Lecture Notes in Computer Science., Springer (2005) 339–353
5. Teodosiu, D., Bjørner, N., Gurevich, Y., Manasse, M., Porkka, J.: Optimizing file replication over limited-bandwidth networks using remote differential compression. Technical report, Microsoft Research (November 2006) MSR-TR-2006-157.
6. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM **21**(7) (1978) 558–565
7. Baquero, C., Moura, F.: Improving causality logging in mobile computing networks. ACM Mobile Computing and Communications Review **2**(4) (October 1998) 62–66
8. Kang, B., Wilensky, R., Kubiatowicz, J.: The hash history approach for reconciling mutual inconsistency. In: ICDCS, IEEE Computer Society (2003) 670–677
9. Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Towards a Tool Environment for Model-Based Testing with AsmL. In Petrenko, A., Ulrich, A., eds.: FATES. Volume 2931 of Lecture Notes in Computer Science., Springer (2003) 252–266
10. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In Alur, R., Peled, D., eds.: CAV. Volume 3114 of Lecture Notes in Computer Science., Springer (2004) 484–487
11. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. Software Eng. **23**(5) (1997) 279–295
12. Jr., T.W.P., Guy, R.G., Heidemann, J.S., Popek, G.J., Mak, W., Rothmeier, D.: Management of Replicated Volume Location Data in the Ficus Replicated File System. In: USENIX Summer. (1991) 17–30
13. Allchin, J.E.: A Suite of Robust Algorithms For Maintaining Replicated Data Using Weak Consistency Conditions. In: Symposium on Reliability in Distributed Software and Database Systems. (1983) 47–56
14. Fischer, M.J., Michael, A.: Sacrificing Serializability to Attain High Availability of Data. In: PODS, ACM (1982) 70–75
15. Oppen, D.C., Dalal, Y.K.: The clearinghouse: A decentralized agent for locating named objects in a distributed environment. ACM Trans. Inf. Syst. **1**(3) (1983) 230–253
16. Lin, S., Pan, A., Guo, R., Zhang, Z.: Simulating Large-Scale P2P Systems with the WiDS Toolkit. In: MASCOTS, IEEE Computer Society (2005) 415–424