Verifying file systems with ACL2

Towards verifying data recovery tools

Mihir P. Mehta
University of Texas at Austin
Austin, TX, USA
mihir@cs.utexas.edu

ACM Reference format:
Mihir P. Mehta. 2016. Verifying file systems with ACL2. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 3 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this paper, we describe work in progress to model and verify filesystems using the ACL2 theorem prover.

2 MOTIVATION

Filesystems are ubiquitous, and a critical factor in the security and performance of all applications. Yet, they remain poorly understood, a problem which has been exacerbated by the complexity of modern filesystems which use redundancy and caching in order to be faster and more reliable. As a consequence, many tools which interact deeply with the filesystem, such as file deletion and file recovery tools, have become more vulnerable to bugs because of the complexity of these tasks. Thus, it is worthwhile to work towards formally verifying the guarantees provided by a filesystem.

3 MODELLING A FILESYSTEM

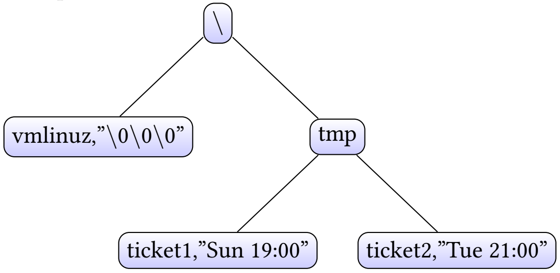
In order to make our proofs of correctness tractable, we choose to make several verified filesystem models in increasing order of complexity. This approach supports incremental proof strategies, providing us with a choice between proving a model equivalent to the next, and simply adapting existing proofs for the next model.

While starting out, we faced a decision about the file system operations we should provide. Following the example of the Google File System [?], we decided to restrict ourselves to a small number of fundamental file system operations - namely reading, writing, creating, and deleting a file. This excludes the operations of opening and closing a file; we hope to implement these when they become necessary for verification in a multiprogramming environment.

4 MODEL 1

The intuitive mental model of a filesystem is a directory tree, which remains useful even though it fails for filesystems with links. Accordingly, it is appropriate for our first model, which will serve as a base specification for all later models, to be a literal tree. Our filesystem recogniser, `l1-fs-p`, recognises trees where each leaf node is either a regular file or an empty directory, and each non-leaf node is a directory containing one or more regular files and subdirectories.

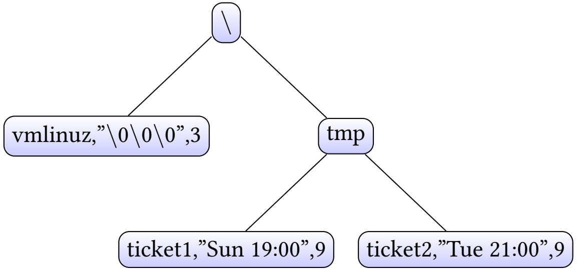
Below, we include a sample of a filesystem tree that is recognised by l1-fs-p.



5 MODEL 2

Model 1 can hold unbounded text files and nested directory structures. However, real filesystems include metadata, and including metadata in our filesystem representation also allows us to define a notion of ”consistency” wherein the actual contents of a regular or directory file are checked for agreement with the metadata. Thus, in our next model, we add an extra field for length of a regular file. We also create a simple version of fsck that checks file contents for consistency with the stated length, and verify that the operations for writing, creating and deleting preserve this notion of consistency.

Below, we include a sample of a filesystem tree that is recognised by l2-fs-p, and a code listing.



```
(DEFUN
  L2−FS−P (FS)
  (DECLARE (XARGS :GUARD T))
  (IF
   (ATOM FS)
   (NULL FS)
```

This work is supported by a grant from the NSF. .

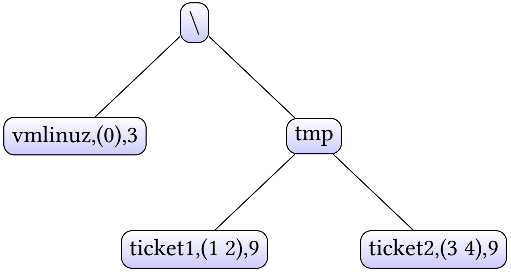
Conference'17, Washington, DC, USA
© 2016 ACM. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

```
   (AND
    (LET
     ((DIRECTORY-OR-FILE-ENTRY (CAR FS)))
     (IF (ATOM DIRECTORY-OR-FILE-ENTRY)
         NIL
         (LET
          ((NAME (CAR DIRECTORY-OR-FILE-ENTRY))
           (ENTRY (CDR DIRECTORY-OR-FILE-ENTRY)))
          (AND (SYMBOLP NAME)
               (OR (AND (CONSP ENTRY)
                        (STRINGP (CAR ENTRY))
                        (NATP (CDR ENTRY)))
                   (L2-FS-P ENTRY))))))
        (L2-FS-P (CDR FS)))))
```

6 MODEL 3

Next, we would like to move towards a more realistic file storage paradigm where the contents of a regular file are broken into fixed-size blocks and stored in an external table, which we will refer to as the disk. In this model, we store the text of a regular file in the disk, and retain only the indices of the relevant blocks in the filesystem tree. For now, we consider the disk to be unbounded and make no attempt at garbage collection. Thus, file creation and writing operations can be represented as append operations, where the new blocks representing the new contents of a file are simply placed at the end of the disk with no effort to free the old blocks or erase their contents. Similarly, deleting a file does not require any disk operations; the blocks of such a file remain in the disk but are no longer referred to.

As before, we include a sample of a filesystem tree that is recognised by l3-fs-p and a code listing.



```
(DEFUN L3-REGULAR-FILE-ENTRY-P (ENTRY)
  (DECLARE (XARGS :GUARD T))
  (AND (CONSP ENTRY)
       (NAT-LISTP (CAR ENTRY))
       (NATP (CDR ENTRY))
       (FEASIBLE-FILE-LENGTH-P (LEN (CAR ENTRY))
                               (CDR ENTRY))))

(DEFUN
  L3-FS-P (FS)
  (DECLARE (XARGS :GUARD T))
  (IF
   (ATOM FS)
   (NULL FS)
   (AND
    (LET
     ((DIRECTORY-OR-FILE-ENTRY (CAR FS)))
     (IF (ATOM DIRECTORY-OR-FILE-ENTRY)
         NIL
         (LET
          ((NAME (CAR DIRECTORY-OR-FILE-ENTRY))
           (ENTRY (CDR DIRECTORY-OR-FILE-ENTRY)))
          (AND (SYMBOLP NAME)
               (OR (L3-REGULAR-FILE-ENTRY-P ENTRY)
                   (L3-FS-P ENTRY))))))
    (L3-FS-P (CDR FS)))))
```

7 MODEL 4

In this model, we finitise our disk; this necessitates garbage collection which we approximate through reference counting. Since we allow neither symbolic links nor hard links in our filesystem, the reference count of any block in the disk is either 0 or 1. This allows us to implement reference counting through an allocation vector, i.e. an array of booleans with the same length as the disk. Thus, in every write or delete operation, the allocation vector entries corresponding to blocks which are no longer used must be marked free; similarly, in every write or create operation, the allocation vector must be scanned to find the appropriate number of free blocks. The lockstep updates described here allow us to prove that aliasing between different files does not occur.

The recogniser l4-fs-p is defined to be the same as l3-fs-p, which makes our equivalence proofs simpler. This arises from the fact that reference counting does not require any changes in the filesystem tree or the disk. At the time of writing, we are in the process of proving equivalence between model 4 and model 3; towards that end, we have proved uniqueness and disjointness properties that ensure our file update operations do not ever alias disk blocks in such a way that they are referred to by two different files, or twice by the same file.

8 PROOF APPROACH

Initially, we would like to prove two well-known properties from the first-order theory of arrays, adapted to the filesystem context. These are the well-known read-over-write properties, which show the integrity of the filesystem.

1. Reading from a location after writing to the same location should yield the data that was written.
2. Reading from a location after writing to a different location should yield the same result as reading before the write.

While these properties are simple enough to state, proving them turns out to be surprisingly subtle. As a point of reference, proving these properties for l1, our initial model, required us to manually specify an induction scheme with 6 conditional branches. As noted before, we have modelled our filesystem incrementally in order to make our proofs tractable, thus, in each successive model, we prove a theorem showing the model to be equivalent to the previous one.

For instance, we define the following function for transforming instances of model 2 to model 1.

```
(DEFUN L2−TO−L1−FS (FS)
  (DECLARE (XARGS :GUARD (L2−FS−P FS)))
  (IF (ATOM FS)
      FS
      (CONS
       (LET*
        ((DIRECTORY−OR−FILE−ENTRY (CAR FS))
         (NAME (CAR DIRECTORY−OR−FILE−ENTRY))
         (ENTRY (CDR DIRECTORY−OR−FILE−ENTRY)))
        (CONS NAME
              (IF (AND (CONSP ENTRY)
                       (STRINGP (CAR ENTRY)))
                  (CAR ENTRY)
                  (L2−TO−L1−FS ENTRY))))
       (L2−TO−L1−FS (CDR FS)))))
```

Then, we can prove the implementation of the write operation in model 2 correct with respect to the specification of model 1 by proving the property illustrated below.

$$\begin{tikzcd}
l2 \arrow[r, "\text{write}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & l2 \arrow[d, "\text{l2-to-l1-fs}"'] \\
l1 \arrow[r, "\text{write}"'] & l1
\end{tikzcd}$$

Similarly, we can prove the implementation of the read operation in model 2 correct with respect to the spec in model 1.

$$\begin{tikzcd}
l2 \arrow[r, "\text{read}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & text \\
l1 \arrow[ur, "\text{read}"'] &
\end{tikzcd}$$

Combining these proofs as shown below, we are able to prove the read-after-write properties for model 2 based on our proof for model 1.

$$\begin{tikzcd}
l2 \arrow[r, "\text{write}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & l2 \arrow[r, "\text{read}"'] \arrow[d, "\text{l2-to-l1-fs}"'] & text \\
l1 \arrow[r, "\text{write}"'] & l1 \arrow[ur, "\text{read}"'] &
\end{tikzcd}$$

9 FUTURE WORK

As previously mentioned, we would like to add the system calls open and close with the introduction of file descriptors. This would be a step towards the study of concurrent FS operations. We would also like to linearise the tree, leaving only the disk - this would be more in keeping with realistic file systems that do not require an in-memory tree representation, but still allow tree traversal through systematic lookups in the disk.

Eventually, we would like to emulate the CP/M filesystem as a convincing proof of concept. This would be a step towards verified versions of fsck and file recovery tools, which could be based on our proofs about the underlying filesystem.

10 RELATED WORK

Currently, the state of the art is represented by Haogang Chen's dissertation work [?], in which the author uses Coq to build a filesystem (named FSCQ) which is proven safe against crashes. This implementation was exported into Haskell, and showed comparable performance to ext4 when run on the Linux kernel through the FUSE layer.

Our work takes a different approach - our aim is to produce verified models of existing filesystems that have binary compatibility with the filesystem layout read and written by the corresponding implementation. This allows us to find bugs in existing filesystems, which is not addressed by Chen's work.

11 CONCLUSION

Through this work, we have gone into some depth on the nuts and bolts of implementing a filesystem at the byte level. In the process, we have demonstrated ACL2's capability to deal with systems-level problems in addition to the hardware verification problems to which it has traditionally been applied.

12 OBTAINING THE CODE

This work is hosted on GitHub, under the GPL 3.0 licence. The code repository can be cloned anonymously using the HTTPS URL https://github.com/airbornemihir/turbo-octo-sniffle.git, and the repository itself can be viewed at https://github.com/airbornemihir/turbo-octo-sniffle.