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),* 5 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. We decided against implementing the entirety of the Linux VFS interface[5], reasoning that this would require us to implement 19 inode operations, 6 dentry operations and 22 file operations. Following the example of the Google File System [4], 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.

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

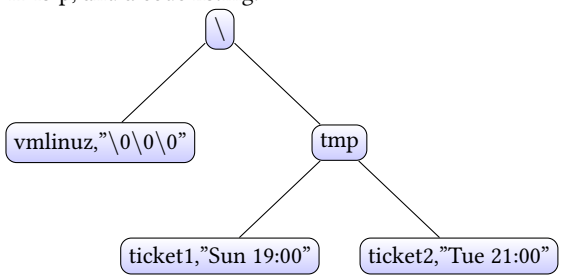
Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

4 MODEL 1

The intuitive mental model of a filesystem is a 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 specification for all later models, to be a literal tree. Our filesystem recogniser, `l1-fs-p`, recognises `symbol-alists` where each cdr of a pair in the alist satisfies either `stringp` (denoting a regular file) or `l1-fs-p` (denoting a subdirectory).

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



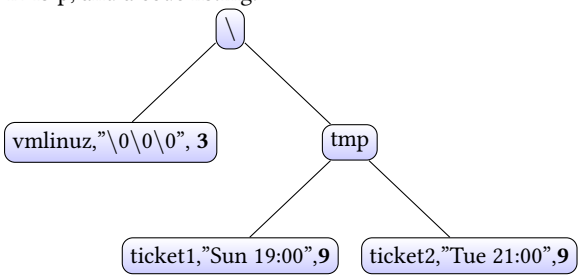

```
(DEFUN
  L1-FS-P (FS)
  (DECLARE (XARGS :GUARD T))
  (IF
   (ATOM FS)
   (NULL FS)
   (AND
    (LET ((DIRECTORY-OR-FILE-ENTRY (CAR FS)))
         (IF (ATOM DIRECTORY-OR-FILE-ENTRY)
             NIL
             (LET
              ((NAME (CAR DIRECTORY-OR-FILE-ENTRY))
               (ENTRY (CDR DIRECTORY-OR-FILE-ENTRY)))
              (AND (SYMBOLP NAME)
                   (OR (STRINGP ENTRY)
                       (L1-FS-P ENTRY))))))
    (L1-FS-P (CDR FS)))))
```

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.

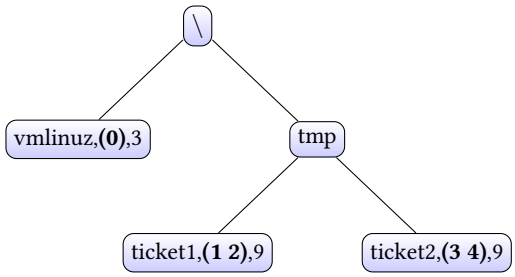



```
(DEFUN
  L2-FS-P (FS)
  (DECLARE (XARGS :GUARD T))
  (IF
   (ATOM FS)
   (NULL FS)
   (AND
    (LET
     ((DIRECTORY-OR-FILE-ENTRY (CAR FS)))
     (IF (ATOM DIRECTORY-OR-FILE-ENTRY)
         NIL
         (LET
          ((NAME (CAR DIRECTORY-OR-FILE-ENTRY))
           (ENTRY (CDR DIRECTORY-OR-FILE-ENTRY)))
          (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

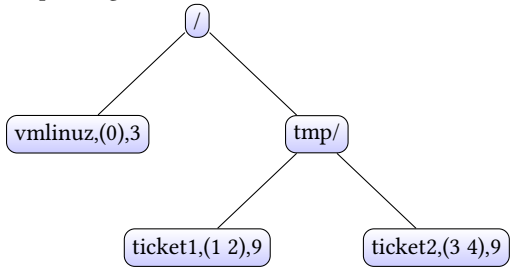
In this model, we finitise our disk; this necessitates garbage collection which we approximate through reference counting. Since we allow neither symbolic links nor hard links in our filesystem, the reference count of any block in the disk is either 0 or 1. This allows us to implement reference counting 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

Table 1: Model 4 - disk and allocation vector

0	\0\0\0	true
1	Sun 19:0	true
2	0	true
3	Tue 21:0	true
4	0	true
5		false

that reference counting does not require any changes in the filesystem tree or the disk. A sample filesystem tree follows, with the corresponding disk and allocation in table 1.



An interesting insight from the proofs of correctness for model 4 was that there is no simple transformation from instances of model 4 to instances of model 3. The contents of a single file are always contiguous in model 3 but not in model 4, and this makes it difficult to derive a transformation that will support a read-over-write proof. Thus, we chose a different strategy: defining a transformation l4-to-l2-fs from instances of l4 to instances of model 2. Thanks to the common definition of l4-fs-p and l3-fs-p, we were able to reuse the already defined l3-to-l2-fs here, which again simplified our proof effort.

8 MODEL 5

This model extends model 4 with the addition of a new kind of metadata, namely file permissions. Incorporating read/write permissions for the user and others, this model develops the necessary infrastructure for more complex permissions checking mechanisms, including user groups and access control lists. We opted to keep the block allocation and garbage collection algorithms the same; this made it easier to prove an equivalence between model 5 and model 4, and in turn, the read-over-write theorems.

No new data structures are added to this model - only metadata for file permissions and ownership. However, this is the first model for which we added getter and setter functions for the data and metadata in a regular file, allowing us to keep the recogniser for a regular file disabled and simplify our proof effort.

Below, we provide a code listing for l5-fs-p and l5-regular-file-entry-p.

```
(DEFUND L5-REGULAR-FILE-ENTRY-P (ENTRY)
  (DECLARE (XARGS :GUARD T))
  (AND (EQUAL (LEN ENTRY) 6)
       (NAT-LISTP (CAR ENTRY))
       (NATP (CADR ENTRY))
       (FEASIBLE-FILE-LENGTH-P (LEN (CAR ENTRY)) (CADR ENTRY))
       (BOOLEANP (CAR (CDDR ENTRY)))
       (BOOLEANP (CADR (CDDR ENTRY)))
       (BOOLEANP (CAR (CDDR (CDDR ENTRY))))
       (BOOLEANP (CADR (CDDR (CDDR ENTRY))))
       (NATP (CDDR (CDDR (CDDR ENTRY))))))

(DEFUN L5-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 (L5-REGULAR-FILE-ENTRY-P ENTRY)
                        (L5-FS-P ENTRY))))))
         (L5-FS-P (CDR FS)))))
```

9 MODEL 6

Next, we take another step towards modelling a FAT32 filesystem by replacing the allocation vector with a file allocation table. We choose to keep the same block allocation and garbage collection mechanisms, which is allowed by the FAT32 specification[]. However, this change creates a number of proof challenges, arising from the fact that the reading a regular file of length n now requires $n/8 - 1$ lookups in the file allocation table, compared to the previous models (4, and 5) where no lookups in the allocation vector were required for the read operation. Thus, a regular file in our model is now susceptible to cycles in the file allocation table, and we are obliged to provide a file read semantics that makes sense in scenarios like this one. We choose to skip error codes for now, and provide a "completion semantics" of returning strings of the requested length for a read even in the scenario of a file length mismatch or a cycle.

10 PROOF APPROACH

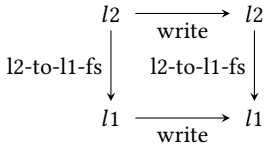
Initially, we would like to prove two properties from the first-order theory of arrays, adapted to the filesystem context. These are the well-known read-over-write properties, which show the integrity of the filesystem.

(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.

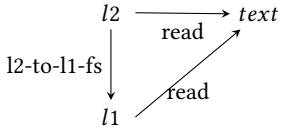
While these properties are simple enough to state, proving them turns out to be surprisingly subtle. As a point of reference, proving these properties for l1, our initial model, required us to manually specify an induction scheme with 6 conditional branches. As noted before, we have modelled our filesystem 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)))))
```

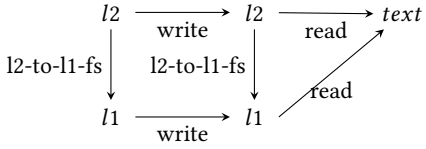
Then, we can prove the implementation of the write operation in model 2 correct with respect to the specification of model 1 by proving the property illustrated below.



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



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.



11 EVALUATION

At present, the codebase spans 6017 lines of ACL2 code, including 118 function definitions and 419 defthm events. Some of this data was obtained by David A. Wheeler's sloccount tool.

In table 2 we note the time taken to certify the models in ACL2, as well as some infrastructure upon which the models are built.

Additionally, for model 5, we ran some tests involving writing a string to a file in the filesystem. The results are shown in table 3.

Table 2: Time taken to prove models

Model 1	1.167s
Model 2	5.672s
Model 3	7.628s
Model 4	31.045s
Model 5	19.925s
Miscellaneous shared lemmas	2.577s

Table 3: Time taken to write strings

Length of string	Time taken
4096	0.05s
16384	0.10s
65536	0.34s
262144	1.55s
1048576	10.75s

12 FUTURE WORK

Having incorporated garbage collection and metadata into the filesystem model, the next challenge is the linearisation of the filesystem model. 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.

We are also planning 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. An alternative approach would be to model concurrent operations after the fashion of Sun NFS [10]; this is also under consideration.

Eventually, we would like to emulate the FAT32 filesystem. This would be a step towards verified versions of fsck and file recovery tools, which would be based on our proofs about the underlying filesystem.

13 RELATED WORK

Fileystem verification is a research topic of long standing. An early effort is [2], where the authors opted to use an interactive theorem prover (Athena) to prove a simulation relation between their implementation and a specification of their own creation.

A landmark paper in the kernel verification space is [7]. The authors, here, took a microkernel approach, and as a consequence filesystem operations do not appear in the kernel they verified. They take up the problem later in [6], creating and verifying a new high-performance filesystem named BilbyFS, and in [1], presenting a new framework to assist systems programmers with write verifiable filesystems.

Currently, the state of the art is represented by Haogang Chen's dissertation work [3], 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.

Another recent work in this space, Hyperkernel [8] is a "push-button" kernel verification effort using the Z3 SMT solver. However, in order to accommodate the limitations of Z3, Hyperkernel approximates by changing POSIX system calls for ease of verification.

Our work takes a different approach - our aim is to produce verified models of existing filesystems that are *executable*, with *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. [9] is a paper featuring an executable specification, but not binary compatibility, which remains an unsolved problem.

14 CONCLUSION

Through this work, we have gone into some depth on an approach towards implementing a filesystem with several essential features (block allocation, file-level metadata, garbage collection) found in real filesystems. 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.

15 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.

REFERENCES

[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, and others. 2016. Cogent: Verifying high-assurance file system implementations. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 175–188.

[2] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. 2004. Verifying a file system implementation. In *International Conference on Formal Engineering Methods*. Springer, 373–390.

[3] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2016. Using Crash Hoare Logic for Certifying the FSCQ File System. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association. https://www.usenix.org/conference/atc16/technical-sessions/presentation/chen_haogang

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 29–43.

[5] Michael K Johnson. 1996. A tour of the Linux VFS. (1996). http://www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html

[6] Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. 2014. File systems deserve verification too! *ACM SIGOPS Operating Systems Review* 48, 1 (2014), 58–64.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, and others. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

[8] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 252–269. DOI: http://dx.doi.org/10.1145/3132747.3132748

[9] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 38–53.

[10] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. 1985. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX conference*. 119–130.