Binary-Compatible Verification of Filesystems with ACL2

Mihir Parang Mehta
University of Texas at Austin, USA
`http://www.cs.utexas.edu/users/mihir`
mihir@cs.utexas.edu

William R. Cook
University of Texas at Austin, USA
`http://www.cs.utexas.edu/users/wcook`
wcook@cs.utexas.edu

Abstract

Filesystems are an essential component of most computer systems. Work on the verification of filesystem functionality has been focused on constructing new filesystems in a manner which simplifies the process of verifying them against specifications. This leaves open the question of whether filesystems already in use are correct at the binary level.

This paper introduces LoFAT, a model of the `FAT32` filesystem which efficiently implements a subset of the `POSIX` filesystem operations, and HiFAT, a more abstract model of `FAT32` which is simpler to reason about. LoFAT is proved to be correct in terms of refinement of HiFAT, and made executable by enabling the state of the model to be written to and read from `FAT32` disk images. EqFAT, an equivalence relation for disk images, considers whether two disk images contain the same directory tree modulo reordering of files and implementation-level details regarding cluster allocation. A suite of co-simulation tests uses EqFAT to compare the operation of existing `FAT32` implementations to LoFAT and check the correctness of existing implementations of `FAT32` such as the `mtools` suite of programs and the Linux `FAT32` implementation. All models and proofs are formalized and mechanically verified in ACL2.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases interactive theorem proving, filesystems

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.25

Related Version A preprint version was published at `https://easychair.org/publications/preprint/dMh7`.

Supplement Material The proof development described in this paper has been incorporated into the ACL2 Community books; these are part of the ACL2 distribution on GitHub (`http://www.github.com/acl2/acl2`). The source code for the models and proofs, with instructions for certifying the models, is available (`https://github.com/acl2/acl2/tree/master/books/projects/filesystems`).

Funding *Mihir Parang Mehta*: This material is based upon work supported by the National Science Foundation under Grant No. CNS-1525472.

Acknowledgements Thanks to Warren A. Hunt Jr. and Matt Kaufmann for their guidance.

1 Introduction

Filesystems offer a critical part of the functionality of modern operating systems, going beyond the basic functionality of persistent storage to offer crash consistency, concurrent data access, and distributed operation. Within the formal methods community, filesystem verification is becoming a mature discipline with the development of high-performance filesystems accompanied by proofs of increasingly expansive notions of correctness. However,

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach; Article No. 25; pp. 25:1–25:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

it is often necessary to verify the operation of an existing filesystem which is known to be suitable in a particular context, in terms of properties such as CPU usage, memory usage, or fragmentation behavior. This remains a challenge.

This paper shows the construction of an executable model of the `FAT32` filesystem, using the interactive theorem prover ACL2 [23], which is useful for reasoning about programs that interact with the filesystem. The aim for this effort is *binary compatibility*, i.e. byte-level correspondence between the model and existing, mature implementations of `FAT32`. This is achieved through a careful examination of the specification of `FAT32` and the behavior of its implementations. Binary compatibility enables reasoning at a low level of abstraction about the precise sequences of bytes accepted and returned by `POSIX` system calls, as well as their return values and the `errno` [24] values set by them. By building this model, LoFAT, incrementally in the refinement style, we are able to address these low-level details while adhering to a more abstract model, HiFAT, which is easier to reason about. The refinement relation between LoFAT and HiFAT is proved.

LoFAT is executable; it includes functionality to read the filesystem state from and write the filesystem state to `FAT32` disk images. The disk image is a convenient abstraction to represent the state of the filesystem, and by interacting directly with disk images the verified implementation needs to trust only a small number of ACL2 functions for writing and reading. Optimization of these procedures for faster I/O enables the efficient execution of the model and co-simulation with existing implementations of `FAT32` over various types of file operations, which helps find bugs.

We begin by providing some necessary details about the reasoning and execution properties of the ACL2 theorem proving system (Section 2). Touching on the `FAT32` filesystem's on-disk format, we proceed to introduce LoFAT and HiFAT[1] (Section 3), detailing the refinement relation between these models and the proof thereof (Section 4). We examine some performance considerations involved in making executions of LoFAT efficient enough for co-simulation tests, and describe the co-simulation tests developed (Section 5). We briefly review the related work (Section 6) and outline some plans for concurrency and crash consistency-related future extensions of this work (Section 7).

2 Background on ACL2

The ACL2 theorem proving system consists of a language, which is a pure functional subset of Common Lisp, and a prover which discharges proof obligations expressed in this language.[2] ACL2, employing an untyped first-order logic, incorporates many automated strategies for discharging first-order goals while also allowing user control of the proof process at different levels of abstraction. As in mathematics, the proof of a conjecture in ACL2 usually relies on the proof of simpler lemmas (*rules*). Most often, these lemmas are *rewrite rules* for rewriting a certain type of term under certain hypotheses; however, other types of rules exist, such as linear rules for arithmetic reasoning.

2.1 Guard verification

A function can optionally have a *guard*, an arbitrary propositional formula in terms of its arguments which is checked to be true at runtime. ACL2 generates a proof obligation stating that the guards of all functions called within the function body are satisfied when the guard

[1] These names respectively refer to Low Level of Abstraction and High Level of Abstraction.
[2] In the literature, the term ACL2 is sometimes used to refer to the language, and sometimes used to refer to the prover.

itself is satisfied. The proof of this obligation is optional; when a function is not guard-verified, guards for function calls within the body are instead checked at runtime. Guard-verified functions, however, avoid these runtime checks, and in general execute faster because guards often include constraints on the type of the function's arguments which allow space to be efficiently allocated for fixed-width integers, strings, and the like. Guard verification helps correct programming errors early and often leads to the formulation of lemmas which can be reused in later proofs.

The guard mechanism also supports `mbe` (*must be equal*) [4], an ACL2 construct which allows the user to locally decouple logical meaning and operational behavior. In a function body, a sub-expression of the form `(mbe :logic term1 :exec term2)` will be treated as meaning `term1` during reasoning but will behave as `term2` at runtime; this enables optimization by a choice of `term2` which is efficient during execution. This is sound because `mbe` extends the function's guard obligation to include the statement that `term1` and `term2` are equal in their local context when the function's guard is satisfied.

2.2 Single-threaded objects

In applicative settings, updates to data structures result in the creation of a new copy of the data structure, which can prove expensive in terms of time and memory. In ACL2, this kind of performance penalty is avoided by the use of immutable data structures called *single-threaded objects*, or stobjs [9]. Stobjs are aggregate structures with scalar and array fields, equipped with the usual applicative semantics, but restricted syntactically to ensure that only one copy of the stobj can be referenced at a given time. With just one immutable copy of the stobj, accesses and updates to scalar and array fields can be implemented in constant time.

As with all aggregate data structures, proving invariants of algorithms involving stobjs necessitates lemmas about the invariance of stobj fields while updating other fields (akin to *frame axioms* [40], although these lemmas are not axioms of the theory). ACL2 macros are used to reduce the effort required to generate these lemmas.

2.3 Equivalence and rewriting

In ACL2, binary predicates can be proved to be equivalence relations. Such an equivalence is treated like first-order equality, in that rules can be formulated to rewrite terms in the context of the given equivalence.

We use a few standard techniques for defining and establishing equivalences in ACL2's untyped logic.

- When a subset relation can be defined on objects which are to be assigned to equivalence classes, equivalent objects can be defined to be subsets of each other. Then, the proofs of reflexivity, symmetry and transitivity arise from the proofs of reflexivity, anti-symmetry and transitivity for the subset relation.
- When a transformation exists between two types, objects of the first type can be defined to be equivalent when they transform to the same object (modulo a previously defined equivalence) of the second type.
- Sometimes an equivalence relation needs to be defined on some notion of well-formed objects (such as objects which can be transformed to objects of a different type). However, guards notwithstanding, all functions in ACL2 must be total including equivalence predicates. In such a case, the predicate can be made a total function by assigning all

ill-formed objects to the same equivalence class and assigning no well-formed objects to this class. This renders the claim of reflexivity, symmetry and transitivity trivial in the ill-formed case.

2.4 Logical story of I/O

Theorem proving systems generally have interfaces with the operating system which are unverified, because operating-system activity is unpredictable and may not return a consistent result on two calls to the same function with the same arguments. This is also the case with ACL2, which provides I/O functionality at various levels of abstraction for programmer convenience. However, a *logical story of I/O* [13] is adhered to, consisting of formal specifications for these I/O functions in terms of their input/output behavior and errors passed on from the operating system. These formal specifications exist in the ACL2 logic and support proofs about sequences of I/O operations and optimizations thereof.

3 FAT32 – specification and modeling

FAT32 was previously the default filesystem for the Windows operating system, and continues to see widespread use in embedded systems and in removable media.

Having detailed the data organization of a FAT32 disk image in our earlier work [34], we limit ourselves here to a brief summary of the on-disk data structures, i.e. the *reserved area*, the *file allocation table*, and the *data region*. Unless otherwise specified, we refer to both regular files (which contain sequences of bytes) and directory files (which contain sequences of directory entries pointing to other files with names, access times and other metadata for each) as *files*.

- The contents of all files are split into fixed-size *clusters* (or extents); these clusters are stored in the data region.
- Linked lists, called *clusterchains*, yield the sequences of clusters belonging to a given file; these clusterchains are stored in the file allocation table. Multiple copies of the file allocation table are allowed in order to protect against data loss in the event of corruption; however only the first one is considered authoritative, and a FAT32 implementation may update the redundant copies infrequently (or not at all).
- The reserved area is a collection of scalar and array fields which specify such volume-wide metadata for the filesystem as the location of the root directory, the size of a cluster, and the number of clusters.

Microsoft provides an authoritative FAT32 specification [35], which includes a number of constraints on the various scalar and array fields, specifying such things as the maximum and minimum number of clusters, the maximum sizes of regular and directory files, and the allowable sizes of clusters. It is necessary to incorporate these constraints into our formal development in order to reason about upper bounds on the sizes of the data structures we allocate and avoid impossible corner cases while proving other useful properties.

Thus, to define our model LoFAT, we first define a single-threaded object type recognized by the predicate `fat32-in-memoryp`. Augmenting this predicate with clauses for the various FAT32 constraints, we obtain the predicate `lofat-fs-p` (Listing 1), which recognizes valid instances of LoFAT. These constraints are a subset of the constraints actually stipulated for FAT32, chosen to be as small as possible while meeting our proof needs. This helps us avoid unduly restricting the possible co-simulations we can undertake with FAT32 implementations which may not strictly adhere to the specification.

Listing 1 lofat-fs-p.

```
(defun lofat-fs-p (fat32-in-memory)
  (and
   (fat32-in-memoryp fat32-in-memory)
   ;; There must be at least 512 bytes per sector.
   (>= (bpb_bytspersec fat32-in-memory) *ms-min-bytes-per-sector*)
   ;; Each cluster must contain a positive integer number of sectors.
   (>= (bpb_secperclus fat32-in-memory) 1)
   ;; There is a lower bound and an upper bound to the number of
   ;; clusters.
   (>= (count-of-clusters fat32-in-memory) *ms-min-count-of-clusters*)
   (<= (+ *ms-first-data-cluster* (count-of-clusters fat32-in-memory))
       *ms-bad-cluster*)
   ;; The reserved area must span a positive integer number of
   ;; sectors.
   (>= (bpb_rsvdseccnt fat32-in-memory) 1)
   ;; Zero or more redundant copies of the FAT are allowed.
   (>= (bpb_numfats fat32-in-memory) 1)
   ;; The FAT must span a positive integer number of sectors.
   (>= (bpb_fatsz32 fat32-in-memory) 1)
   ;; The root cluster must exist in the addressable part of the file
   ;; allocation table.
   (>= (fat32-entry-mask (bpb_rootclus fat32-in-memory))
       *ms-first-data-cluster*)
   (< (fat32-entry-mask (bpb_rootclus fat32-in-memory))
      (+ *ms-first-data-cluster* (count-of-clusters fat32-in-memory)))
   (<= (+ (count-of-clusters fat32-in-memory) *ms-first-data-cluster*)
       (fat-entry-count fat32-in-memory))
   ;; The cluster size must be a multiple of the size of a directory
   ;; entry.
   (equal (mod (cluster-size fat32-in-memory) *ms-dir-ent-length*) 0)
   (equal (mod *ms-max-dir-size* (cluster-size fat32-in-memory)) 0)
   ;; The data region must be an array of clusters of the appropriate
   ;; length.
   (stobj-cluster-listp-helper fat32-in-memory
                               (data-region-length fat32-in-memory))
   (equal (data-region-length fat32-in-memory)
          (count-of-clusters fat32-in-memory))
   ;; The file allocation table must contain the appropriate number
   ;; of 4-byte-wide entries.
   (equal (* 4 (fat-length fat32-in-memory))
          (* (bpb_fatsz32 fat32-in-memory)
             (bpb_bytspersec fat32-in-memory)))))
```

It has been argued [32] that the axiomatic verification methodology, wherein specific properties of a system are enumerated and proved, is inadequate for systems of any significant complexity, which can only be verified through refinement. Much of the related work [6, 41, 11] opts to prove the correctness of an implemented filesystem through refinement of a specification, demonstrating a *de facto* consensus on this point. Thus, we choose to develop the abstract model HiFAT, and prove that it is *refined without stuttering* [1] by LoFAT. HiFAT instances are directory trees in which the leaf nodes are regular files and the non-leaf nodes are directories. Each node in a directory tree contains the FAT32 directory entry for the corresponding file, and the full contents of each regular file are stored as strings within the tree. Further, these trees are subject to FAT32's constraints – a regular file may be up to $2^{32} - 1$ bytes long; a directory may contain up to $2^{16} - 1$ directory entries; and a directory may not contain duplicate directory entries.

As a result of this refinement relationship, we are able to reason about file operations in terms of operations on directory trees, while implementing them efficiently in a data format that is very close to the actual structure of a FAT32 disk image.

4 Properties proved

Much of the proof effort for this work concerns the correctness of the transformations between the different FAT32 representations used; these transformations are obliged to terminate in a bounded amount of time, be invertible in terms of appropriate equivalence relations, and return the proper error codes. These proofs lead up to the refinement proof showing the correctness of the POSIX system calls implemented for FAT32 (Section 5.2).

4.1 Termination

ACL2 requires each recursive function definition to be accompanied by a proof that it will terminate in a bounded amount of time. Such a proof is accomplished by defining a function-specific *measure* (in many cases, determined automatically by ACL2) and proving that the measure strictly decreases for each recursive call within the function body. However, termination proofs pose a challenge in many applications where pointer chasing is involved [19, 18]. In the context of FAT32, when transforming a LoFAT instance into an HiFAT instance, pointer chasing is necessary both for regular files and directory files, neccesitating some care towards the avoidance of non-terminating computation in both cases without incurring the overhead of general-purpose cycle detection algorithms.

Each file's directory entry contains the index of its first cluster, and its contents are determined by following its clusterchain in the file allocation table and concatenating together the corresponding clusters. This is subject to potential cycles in the clusterchain. These can be mitigated because of the FAT32 stipulation of maximum lengths for regular files and directory files; the measure for the recursion becomes the remaining length of the file, which decreases with each cluster visited in the file allocation table.

For directory files the problem is more involved; since the transformation of a directory on disk to a directory tree involves the recursive transformation of all sub-directories, it is possible for a sub-directory cycle to arise. Consider an ill-formed disk image where the top-level directory `etc` contains an entry for the sub-directory `apt` and `apt` in turn contains a directory entry for `etc`; in this scenario, it is possible for the algorithm to spin over the fictitious sub-directories `/etc/apt/etc`, `/etc/apt/etc/apt`, . . . A loop-stopping criterion is required which accepts all disk images which are free of cycles and returns an error for all disk images with sub-directory cycles. POSIX defines the constant `PATH_MAX` to bound the length

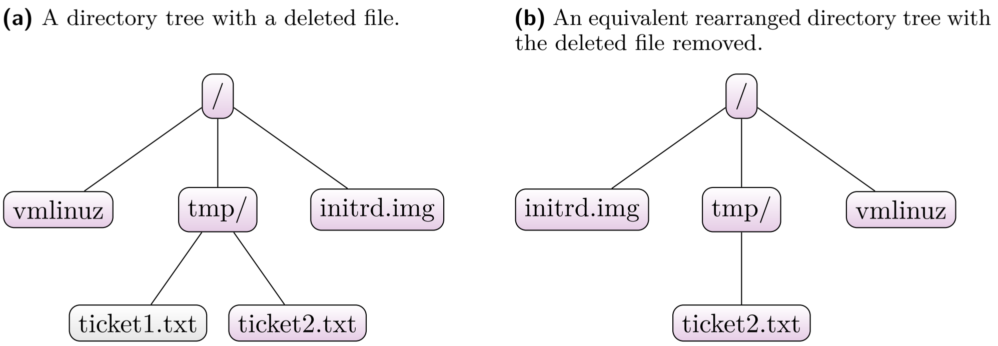

(a) A directory tree with a deleted file.
/
vmlinuz
tmp/
initrd.img
ticket1.txt
ticket2.txt
(b) An equivalent rearranged directory tree with the deleted file removed.
/
initrd.img
tmp/
vmlinuz
ticket2.txt

Figure 1 Two equivalent directory trees.

of a pathname, but it is inconsistently used by implementations [30]; thus a naive solution based on a maximum directory nesting depth is likely to reject valid disk images. A better way is to examine the filesystem at the granularity of directory entries, noting that these cannot exceed the total space available in the data region. Thus, an argument `entry-count` is added to `lofat-to-hifat-helper` (a recursive helper function for the transformation) and designated as the measure, with decrementation for each entry counted when making recursive calls. `entry-count` is instantiated to the maximum number of entries possible in the data region in `lofat-to-hifat` (the top-level wrapper function); this ensures that all valid filesystem instances are accepted without an error and demonstrates the existence of a cycle in each case where the total possible number of directory entries is exceeded.

4.2 Equivalence

Several useful filesystem correctness properties depend, for their proofs, on a notion of equivalence between two filesystem instances. While defining such a notion of equivalence, it is desirable to leave room for different implementation choices for cluster allocation, garbage collection, and other such details. Some constraints which characterize such an equivalence relation follow, and are illustrated in Figure 1.

- Modulo rearrangement, each directory in two equivalent filesystem instances should contain the same regular files and, recursively, the same sub-directories. This ensures that looking up the same pathname in both yields the same results.
- Directory entries for the current directory (`.`) and the parent directory (`..`) should be disregarded, since they do not refer to new unique files. The same is true for deleted files' directory entries.
- Re-allocation of clusters for the contents of a given file without changing the contents should be disregarded.
- Changes to the redundant copies of the file allocation table should be disregarded.
- Changes to volume-level metadata, such the size of a cluster or the total number of clusters in the filesystem, should be taken into account only if they result in the deletion of file data.

Also, to simplify the verification task, creation times, access times, write times, and long names for files are set aside, even though this limits the reasoning which can be carried out about programs which rely on these for their correct operation, such as the incremental compilation system Make [43].

Listing 2 `hifat-equiv`.

```
(defun hifat-equiv (m1-file-alist1 m1-file-alist2)
  (b* ((m1-file-alist1 (hifat-file-alist-fix m1-file-alist1))
       (m1-file-alist2 (hifat-file-alist-fix m1-file-alist2)))
    (and (hifat-subsetp m1-file-alist1 m1-file-alist2)
         (hifat-subsetp m1-file-alist2 m1-file-alist1))))
```

Listing 3 `lofat-equiv`.

```
(defund-nx lofat-equiv (fat32-in-memory1 fat32-in-memory2)
  (b* (((mv fs1 error-code1) (lofat-to-hifat fat32-in-memory1))
       (good1 (and (lofat-fs-p fat32-in-memory1)
                   (equal error-code1 0)))
       ((mv fs2 error-code2) (lofat-to-hifat fat32-in-memory2))
       (good2 (and (lofat-fs-p fat32-in-memory2)
                   (equal error-code2 0)))
       ((unless (and good1 good2)) (and (not good1) (not good2))))
    (hifat-equiv fs1 fs2)))
```

At HiFAT, the most abstract level, we meet the above requirements by first defining a subset relation `hifat-subsetp`; and then defining the equivalence relation `hifat-equiv` (Listing 2) in terms of subsets as discussed in Section 2.3.

At LoFAT, the next lower level of abstraction, we define the equivalence relation `lofat-equiv` in terms of the transformation between LoFAT and HiFAT, once again grouping ill-formed LoFAT instances (that is, instances which return a non-zero error code when transformed to HiFAT) into the same equivalence class (Listing 3). Finally, we define an equivalence relation for disk images. These are strings, each representing the entire contents of the image. This equivalence relation, EqFAT, groups all ill-formed disk images which cannot be transformed to a valid LoFAT instance into the same equivalence class (Listing 4).[3]

[3] Both these functions are considered *non-executable* in ACL2, because they reference two instances of the stobj `fat32-in-memory` at the same time. They are thus introduced with `defund-nx` [3] instead of the usual `defun` and can only be used for reasoning. These functions also use `b*` [2], an ACL2 extension of the Common Lisp `let*` with a more flexible syntax for let-bindings.

Listing 4 EqFAT.

```
(defund-nx eqfat (str1 str2)
  (b* (((mv fat32-in-memory1 error-code1)
        (string-to-lofat (create-fat32-in-memory) str1))
       (good1 (and (stringp str1) (equal error-code1 0)))
       ((mv fat32-in-memory2 error-code2)
        (string-to-lofat (create-fat32-in-memory) str2))
       (good2 (and (stringp str2) (equal error-code2 0)))
       ((unless (and good1 good2)) (and (not good1) (not good2))))
    (lofat-equiv fat32-in-memory1 fat32-in-memory2)))
```

4.3 Invertibility and error codes

HiFAT instances are directory trees, defined recursively; thus, proofs about HiFAT generally require induction. Many theorems about recursive functions can be automatically proved in ACL2 through inference of induction schemes; however, an induction scheme can also be explicitly designated in order to control the inductive formulation of a theorem. In such an induction scheme, the induction hypothesis can be strengthened or weakened as needed.

Between HiFAT and LoFAT, transformations `hifat-to-lofat` and `lofat-to-hifat` are defined, and must be proved to be inverses of each other under the appropriate equivalence relations. This is a claim in two parts: transforming m_1 to m_2 and back should result in an m_1' related to m_1 by `hifat-equiv`; and transforming m_2 to m_1 and back should result in an m_2' related to m_1 by `lofat-equiv`.

The proof of the first part of this claim (as illustrated in Figure 2a) turns out to also involve error codes; no claims can be made about the invertibility of a transformation if it returns an error. Thus, the proof requires a strengthened induction hypothesis to show in tandem that the error code returned by `lofat-to-hifat` while transforming m_2 back to m_1' is 0 (signifying no error.) This induction is the most complex proof undertaken in this work, since it requires an induction scheme to be defined on functions which interpret binary file formats.

The second part of this claim is true by the definition of `lofat-equiv` and an instantiation of the first claim (Figure 2b).

It is also necessary to prove the correctness of the transformations between instances of LoFAT and `FAT32` disk images (strings). These transformations, `lofat-to-string` and `string-to-lofat`, are proved to be mutual inverses under the equivalence relations `equal` (first-order equality) and EqFAT, respectively. As before, one direction of the claim is proved and then instantiated to prove the other by the definition of EqFAT (Figure 2c).

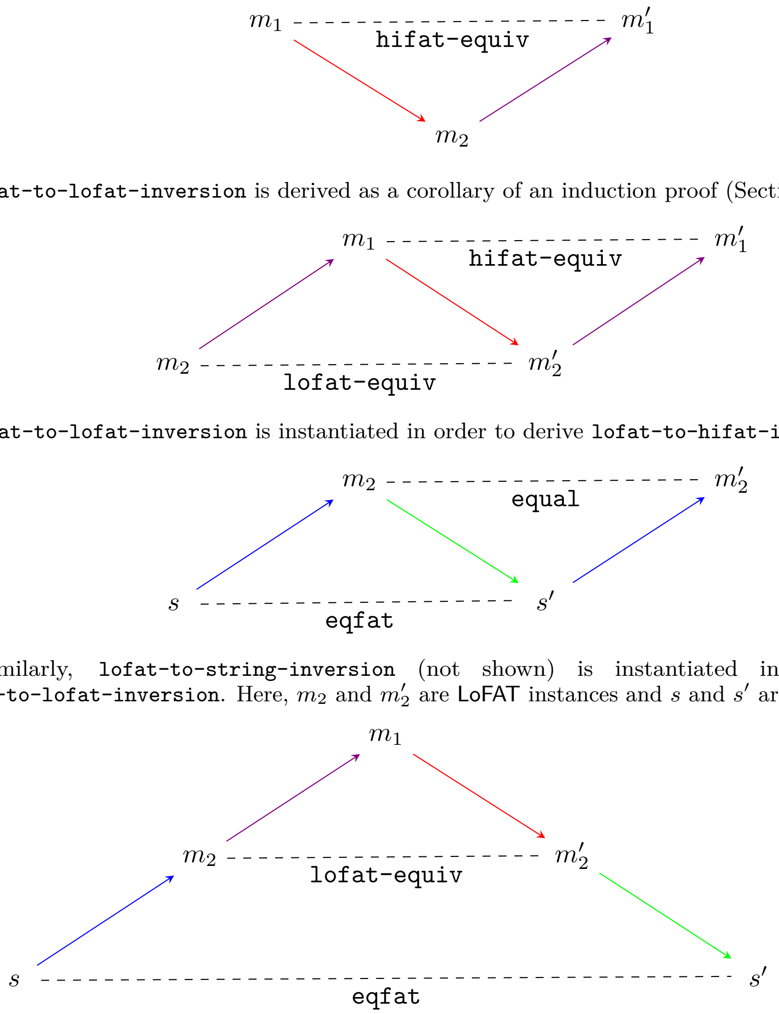
Equality is known to refine all equivalence relations; thus, `equal` refines `lofat-equiv`, and the correctness of the transformations between disk images and HiFAT instances through the intermediate level LoFAT can finally be certified by composing these proofs (Figure 2d).

4.4 Correctness of the specification

Prior filesystem verification work [6] has shown the proof process to uncover subtle bugs in the specification of a filesystem which would otherwise have remained hidden; this has matched our experience modeling and verifying HiFAT and LoFAT. We note some examples of bugs we found in our models in this manner.

- In `FAT32`, the first two entries in the file allocation table are reserved for volume-level metadata; thus, the size of the file allocation table must exceed the number of available clusters by at least two. Additionally, there may be a number of unused entries at the end of the file allocation table, since it must span an integer number of sectors. These differences led us to place incorrect upper bounds on the root cluster of the filesystem, the first cluster of an arbitrary file, and the length of the file allocation table. These errors were discovered and rectified during the proofs of correctness of our transformations.
- An off-by-one bug caused the directory bit of a directory entry to be wrongly set; this was also identified and rectified in the process of proving the transformations correct.

In addition, some bugs in parts of the code which were not immediately verified were found outside of the theorem proving process, by means of co-simulation. One example was the case of a `FAT32` volume in which the root had no directory entries, which is possible in



(a) `hifat-to-lofat-inversion` is derived as a corollary of an induction proof (Section 4.3).

(b) `hifat-to-lofat-inversion` is instantiated in order to derive `lofat-to-hifat-inversion`.

(c) Similarly, `lofat-to-string-inversion` (not shown) is instantiated in order to derive `string-to-lofat-inversion`. Here, m_2 and m_2' are LoFAT instances and s and s' are disk image strings.

(d) `string-to-hifat-inversion` is a corollary of `lofat-to-hifat-inversion`.

Figure 2 Equivalences.

FAT32 because only directories other than the root are required to have . and .. entries. FAT32 constrains each directory file to have at least one cluster, and this constraint had been omitted from the specification. Over a number of co-simulation tests with the Linux FAT32 implementation, this bug was discovered and fixed.

5 Evaluation

5.1 Co-simulation

Co-simulation is a necessary component of formal verification efforts when binary compatibility is the aim, in order to validate the correspondence of the verified model with the software/hardware system in question [17]. A challenge, from the perspective of co-simulation as well as from the perspective of reducing the risk of bugs in unverified code, is the choice of an interface to the operating system. We develop our co-simulation tests as reads and writes on disk images; thus, the potential for bugs outside the verified part of the implementation is confined to specification and implementation errors in ACL2's built-in I/O operations (and indeed, one such bug was found during this development [22]).

Among existing FAT32 implementations, we have chosen to co-simulate with the Linux kernel implementation of FAT32 (as mediated by the GNU Coreutils) and the mtools [31]. The mtools perform various operations such as copying and deletion of files on a given FAT32 disk image or block device, which makes co-simulation relatively straightforward. Co-simulation with the Coreutils involves more steps since they are agnostic towards the underlying filesystem; each test proceeds by mounting a disk image, running the program in question, and unmounting. This co-simulation setup checks the correctness of file operations, without changing filesystem state, in the two following scenarios (which are not mutually exclusive).

1. File operations which retrieve data from the filesystem, such as pread [26], result in output which must be compared to that of the canonical FAT32 implementation. The program diff [15] effects this comparison.
2. File operations which modify the state of the filesystem, such as pwrite [27], result in a modification to the disk image. The modified disk image must then be compared to a disk image modified by the canonical FAT32 implementation; this is done by an ACL2 program which checks whether EqFAT holds for the two images.

5.2 POSIX interface and tests

Table 1 summarizes the subset of the POSIX system calls which have been implemented. The Linux convention is for system calls to return an error code, which is zero if and only if no error occurred, and set the global variable errno; together, these allow an application program making a system call to include error-handling code based on whether an error arose and why. In the ACL2 setting, where there are no global variables, FAT32 system calls maintain the convention by including the "return value" and errno value in the values they return. This matches the Linux implementation of FAT32; thus, for example, when rmdir is called on a non-empty directory, the filesystem instance is returned unmodified along with a non-zero "return value" and an errno value of EEXIST, as specified in the POSIX manual page for rmdir [28]. File descriptors, for operations such as pread and pwrite, are provided through a straightforward implementation of a file table and a file descriptor table, similar to Synergy's [8] implementation; however, the interaction of multiple processes with the filesystem is not yet supported.

This subset suffices for writing and testing ACL2 programs which co-simulate a number of programs from the Coreutils suite (Figure 3a) and from the mtools (Figure 3b). The co-simulation test suite also includes a basic sanity check which compares the output of the program mkfs.fat -v, which creates a FAT32 disk image and prints a textual summary of volume-level metadata [20], with the output of an ACL2 program which reports the same metadata.

For each system call except statfs [29], a version applicable to HiFAT is first developed, and then the LoFAT version is implemented by first transforming the filesystem instance to an HiFAT instance, and then performing the HiFAT version of the system call. If the system call results in a change to the filesystem state, the HiFAT instance is then transformed back to an LoFAT instance at the end. This approach is correct by construction, by the definition of lofat-equiv.

Table 1 `POSIX` syscalls implemented.

Syscall	LoFAT implementation	LoFAT implementation through HiFAT transformation
`close`	✓	✓
`lstat`	✓	✓
`mkdir`		✓
`mknod`		✓
`open`	✓	✓
`pread`	✓	✓
`pwrite`		✓
`rename`		✓
`rmdir`		✓
`statfs`	✓	
`truncate`		✓
`unlink`		✓

Program
`cp`
`ls`
`mkdir`
`mv`
`rm`
`rmdir`
`stat`
`truncate`
`unlink`
`wc`

(a) Coreutils programs co-simulated.

Program
`mcopy`
`mdel`
`mdeltree`
`mmd`
`mmove`
`mrd`
`mren`

(b) `mtools` programs co-simulated.

Figure 3 Syscalls and co-simulation tests.

Co-simulation tests almost always require more than one system call on a given disk image. When this happens, contiguous sequences of operations on the HiFAT instance are carried out while eliding back and forth transformations between HiFAT and LoFAT until the moment of writing back to disk. This elision is sound, as shown by the theorem `lofat-to-hifat-inversion` (Figure 2b), and places HiFAT in a role similar to that of a cache.

`statfs` [29] is an exception and must be implemented at the LoFAT level, since it reports volume-level metadata, such as the total space and free space in the filesystem, which is abstracted away in HiFAT. This also limits the extent to which `statfs`, and programs which use it such as `stat` (more precisely, `stat -f`/`stat --file-system`), can be incorporated into co-simulation tests, because volume-level metadata can differ between filesystems which are identical in terms of the files contained. For instance, the directory tree in Figure 1a contains the same files and directories as the tree in Figure 1b but may still occupy more space on disk, because the directory entry for the deleted file `/tmp/ticket1.txt` still exists and may cause the contents of the directory `/tmp` to occupy an additional cluster.

Since HiFAT is a sparse format for representing the filesystem state, the overheads for these transformations are small enough for co-simulation testing to be feasible; a further improvement in efficiency comes from verifying the guards of all the system calls. However, considering there to be room for improvement in terms of removing these overheads, we also construct provably equivalent implementations of `open`, `pread` and `lstat` for LoFAT which skip the transformation from LoFAT to HiFAT. We are working on doing this for the remaining system calls.

5.3 Performance

This implementation of `FAT32` loads up ACL2 in order to execute the model, which necessarily imposes a lower bound on the time taken for a co-simulation test with a program. However, reasonably quick co-simulation is essential to achieving breadth as well as depth in the co-simulation coverage; thus, optimizations become an important part of the modeling effort. The following two design choices are significant.

1. LoFAT is implemented as a stobj, even though this complicates syntax and reasoning, in order to avoid the performance penalties associated with creating and destroying large immutable data structures each time a single element in the in-memory `FAT32` representation is modified. Transformations to and from HiFAT would have been prohibitive in co-simulation tests without a guard-verified stobj implementation of LoFAT.
2. String representations of data are chosen over byte-list or character-list representations wherever possible. While lists are simpler to reason about, it makes a difference to be able to use the efficient implementations of the built-in string operations `concatenate` (string concatenation), `subseq` (substring extraction) and so on while extracting and reconstructing file contents and working with disk images. In addition, `read-file-into-string` [5], a recent addition to ACL2, provides a fast `mmap`-based [25] alternative to ACL2's character-oriented I/O operations for the use case of reading information from a `FAT32` disk image and populating the fields of the LoFAT instance. Thus, by choosing to work with a string representation for disk images, and by choosing to represent the contents of the data region as an array of cluster-sized strings (Section 3) to take full advantage of the atomicity of clusters in `FAT32`, performance penalties associated with conversions between strings and lists are avoided.

Within the parameters of this design, two optimizations are made possible by ACL2's logical story for I/O operations. Both of these avoid the construction of intermediate string representations while transforming between disk images and LoFAT instances, in order to reduce the associated overheads, while retaining the abstraction of the disk image as a string. Specifically, while writing back to disk, the explicit construction of a data region string would involve an expensive concatenation of all the clusters; this is omitted by instead writing back all the clusters in sequential order. Similarly, while reading a disk image, the population of the data region after having read the disk image would involve multiple `subseq` operations for extracting the clusters, with significant memory allocation overhead; this is avoided by instead calling `read-file-into-string` multiple times with the appropriate offsets to read the pertinent clusters from the disk image directly into the data region of the LoFAT instance. For both these optimizations, `mbe` is used (Section 2.1) to show that the optimized ACL2 code has the same effect. This is in keeping with the refinement style of proof used throughout this work: when a conceptually simple sequence of I/O operations is replaced with a more complex sequence, the simpler sequence is, in a sense, a specification which is refined. This is also how the model development remains tractable as it evolves: while replacing an earlier implementation, in which disk image strings were explicitly handled, with the optimized one, the co-simulation test suite showed the absence of regressions but the proof that both implementations work the same way enabled much greater confidence.

Table 2 Timing disk image I/O.

Disk image size	Read time	Write time
128 MB	2.48 s	4.14 s
256 MB	3.58 s	7.91 s
512 MB	7.52 s	15.46 s
1024 MB	15.92 s	24.87 s

Table 3 Code summary.

Lines of code (models and proofs)	24,905
Lines of code (co-simulation)	619
Co-simulation tests	31

As a result of these design choices and optimizations, co-simulations involving relatively large disk images become possible. For comparison, the in-memory sparse filesystem `tmpfs` [42] usually mounts volumes of size 1 GB to 10 GB on a standard consumer laptop; we have been able to run tests involving disk images of size 1 GB in the same environment. Further, since HiFAT is by nature a sparse format, allocating memory only for file contents, there is little overhead associated with most file operations which affect only the intermediate HiFAT instance. Table 2 lists some timing results for such tests in terms of reading and writing `FAT32` disk images, and Table 3 summarizes statistics pertaining to the magnitude of the modeling effort.[4]

6 Related work

Much of the existing filesystem verification work has taken on the task of synthesizing a new filesystem, developed in a way that simplifies the proofs of filesystem properties of interest.

An early effort was Synergy [8], in which a filesystem was developed and verified according to a specification in ACL2. However, binary compatibility was not a goal for this work, and the design choice of maintaining a mapping from filenames to file contents did not take into account the complexity of path resolution. The `POSIX`-like formulation chosen by Synergy and by other efforts on the more abstract end of the filesystem verification spectrum [16] was an inspiration for an earlier phase of the present work [34], in which `FAT32` models were developed in an incremental fashion from a series of abstract filesystem models, adding more realistic filesystem features in each of the models.

FSCQ [11], developed with Coq [7], is a high-performance FUSE-based [39] filesystem with formally verified crash consistency properties. However, FSCQ exports its executable code to Haskell, and Haskell's FUSE interface to the operating system is, by necessity, unverified. A bug in this interface was discovered through the use of Bounded Black-Box testing, a methodology for automatically testing the data persistence behavior of filesystems [36] and later fixed. FSCQ has been followed by DFSCQ [10], which formally specifies the `fsync` and `fdatasync` file operations, and SFSCQ [21] which proves two-safety confidentiality properties in terms of data noninterference.

COGENT [6], developed with the help of Isabelle/HOL [38], takes a different tack by providing a verified compiler for turning a domain-specific filesystem specification language into verified C code implementing a filesystem.

[4] These statistics were generated using David A. Wheeler's "SLOCCount".

Z3 [14], a non-interactive theorem prover, has also been used for filesystem verification through SMT solving. Hyperkernel [37] attempts a verification of the xv6 [12] microkernel by simplifying it to make the problem tractable through SMT solving. This simplification replaced all kernel data structures with fixed-length implementations, leading all kernel operations (including file operations) to become constant-time. A more filesystem-focused effort is Yggdrasil [41], which verifies a number of filesystem calls by providing a refinement proof showing that its concrete filesystem implementation adheres to a formal specification. This is similar to what FSCQ does, but Yggdrasil's Z3-based verifier achieves this automatically by means of symbolic execution.

7 Future work

While a number of properties have been proved about the FAT32 models and extensive co-simulation tests have been carried out, there remain research questions to be answered in terms of concurrency and generalization to other filesystems.

Within the FAT32 context, a straightforward extension of this work would be to provide an interface closer to that of POSIX, for instance through FUSE [39]. This would allow programs written in C and other languages, which use file descriptors, to interface with our implementation. This, in turn, would help mature the model by facilitating the use of automated testing methodologies such as Bounded Black-Box testing [36] in order to discover more bugs. This would also offer greater opportunities to seek performance gains, including by skipping the transformations to the intermediate HiFAT representation and back in favor of direct manipulation of LoFAT instances or disk images for file operations (as already demonstrated with open) and by using a demand paging-like algorithm to turn LoFAT into a sparse format only storing clusters allocated to files.

We have taken some steps to generalize this work, including the development of macros (Section 2.2), and we are interested in applying this filesystem verification methodology to a binary-compatible verification effort for more complex filesystems with features such as hard linking and crash consistency. The ext4 filesystem [33], which provides crash consistency by means of journaling, is an example.

We are also interested in extending this work to incorporate a model of concurrency, along the same lines as prior work on formalization of microprocessor architectures in theorem proving environments. This would allow the filesystem to serve as a precise specification for correct filesystem behavior in a multiprogramming environment. Making use of such a specification, it would become possible to prove the correctness of programs which concurrently interact with the filesystem and make use of the functionality provided by the operating system to avoid race conditions.

8 Conclusion

A byte-level examination of the specification and existing implementations of a filesystem is a necessary part of a verification effort for it to enable reasoning about the behavior of programs which interact with the filesystem. The recursive definition of the directory tree is central to the study of filesystems; thus, induction is also central to the analysis. Defining and using notions of equivalence between directory trees which disregard implementation details is essential for demonstrating that our FAT32 model and existing FAT32 implementations operate the same way. The logical decoupling enabled by mbe helps keep formal developments involving binary file formats tractable as they evolve through various optimizations, including optimizations based on the logical story of I/O.

This paper's contribution is the general-purpose methodology for binary compatible filesystem verification which makes use of the above techniques and is illustrated through LoFAT and HiFAT. This makes reasonably good performance possible for a disk-image manipulation methodology of verified filesystem implementation, which is sufficient for validating existing filesystem implementations by means of extensive co-simulation testing.

References

1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. doi:10.1016/0304-3975(91)90224-P.

2 ACL2 Community. ACL2 documentation for B*. See URL http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____B_A2.

3 ACL2 Community. ACL2 documentation for DEFUND-NX. See URL http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____DEFUND-NX.

4 ACL2 Community. ACL2 documentation for MBE. See URL http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____MBE.

5 ACL2 Community. ACL2 documentation for READ-FILE-INTO-STRING. See URL http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____READ-FILE-INTO-STRING.

6 Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent: Verifying high-assurance file system implementations. *ACM SIGPLAN Notices*, 51(4):175–188, 2016. doi:10.1145/2872362.2872404.

7 Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013. doi:10.1007/978-3-662-07964-5.

8 William R. Bevier and Richard M. Cohen. An executable model of the Synergy file system. Technical report, Technical Report 121, Computational Logic, Inc, 1996.

9 Robert S. Boyer and J Strother Moore. Single-threaded objects in ACL2. In *International Symposium on Practical Aspects of Declarative Languages*, pages 9–27. Springer, 2002. doi:10.1007/3-540-45587-6_3.

10 Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 270–286. ACM, 2017.

11 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *USENIX Annual Technical Conference*, 2016.

12 Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. URL: http://pdos.csail.mit.edu/6.828/2014/xv6.html.

13 Jared Davis. Reasoning about ACL2 file input. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 117–126. ACM, 2006.

14 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

15 Paul Eggert, Mike Haertel, David Hayes, Richard Stallman, and Len Tower. diff (1)-Linux manual page, accessed: 07 Sep 2018.

16 Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the POSIX file system. In *European Symposium on Programming Languages and Systems*, pages 169–188. Springer, 2014. doi:10.1007/978-3-642-54833-8_10.

17 Shilpi Goel, Warren A. Hunt Jr., Matt Kaufmann, and Soumava Ghosh. Simulation and formal verification of x86 machine-code programs that make system calls. In *Formal Methods*

in Computer-Aided Design (FMCAD), 2014, pages 91–98. IEEE, 2014. doi:10.1109/FMCAD.2014.6987600.

18 David Greve. Address enumeration and reasoning over linear address spaces. In *5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004), Austin, TX*, 2004.

19 David Greve and Matt Wilding. Dynamic datastructures in ACL2: A challenge, 2002. URL: http://hokiepokie.org/docs/festival02.txt.

20 Dave Hudson, Peter Anvin, and Roman Hodek. mkfs.fat (8)-Linux manual page, accessed: 09 Jul 2018.

21 Atalay Ileri, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using DISKSEC. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 323–338, 2018.

22 Matt Kaufmann. Fixed read-file-into-string bug. (commit message), July 2018. URL: https://github.com/acl2/acl2/commit/8388ac10289d5cab791953238057294604af6d60.

23 Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance (COMPASS'96)*, pages 23–34. IEEE, 1996.

24 Michael Kerrisk. errno (3)-Linux manual page, accessed: 07 Sep 2018.

25 Michael Kerrisk. mmap (2)-Linux manual page, accessed: 09 Dec 2018.

26 Michael Kerrisk. pread (2)-Linux manual page, accessed: 09 Jul 2018.

27 Michael Kerrisk. pwrite (2)-Linux manual page, accessed: 09 Jul 2018.

28 Michael Kerrisk. rmdir (2)-Linux manual page, accessed: 17 Mar 2019.

29 Michael Kerrisk. statfs (2)-Linux manual page, accessed: 09 Dec 2018.

30 Evan Klitzke. PATH_MAX is tricky, April 2017. URL: https://eklitzke.org/path-max-is-tricky.

31 Alain Knaff. mtools (1)-Linux manual page, accessed: 09 Dec 2018.

32 Leslie Lamport. Verification and specification of concurrent programs. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 347–374. Springer, 1993. doi:10.1007/3-540-58043-3_23.

33 Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

34 Mihir Parang Mehta. Formalising Filesystems in the ACL2 Theorem Prover: an Application to FAT32. *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and Its Applications*, page 18, 2018.

35 Microsoft. Microsoft Extensible Firmware Initiative FAT32 File System Specification, December 2000. URL: https://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc.

36 Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. *arXiv preprint*, 2018. arXiv:1810.02904.

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

38 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002. doi:10.1007/3-540-45949-9.

39 N. Rath and M. Szeredi. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface, 2018.

40 Murray Shanahan. The Frame Problem. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2016.

41 Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2016.

42 Peter Snyder. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, pages 241–248, 1990.

43 Richard M Stallman. GNU Make, 1988.