Separation Logic-Based Verification atop a Binary-Compatible Filesystem Model

Mihir Parang Mehta, William R Cook

Separation Logic-Based Verification atop a Binary-Compatible Filesystem Model

Mihir Parang Mehta and William R. Cook

University of Texas at Austin, Austin TX 78712, USA
{mihir,wcook}@cs.utexas.edu

Abstract. Filesystems have held the interest of the formal verification community for a while, with several high-performance filesystems constructed with accompanying proofs of correctness. However, the question of verifying an existing filesystem and incorporating filesystem-specific guarantees remains unexplored, leaving those application developers underserved who need to work with a specific filesystem known to be fit for their purpose. In this work, we present an implementation of a verified filesystem which matches the specification of an existing filesystem, and with our new model AbsFAT tie it to the reasoning framework of separation logic in order to verify properties of programs which use the filesystem. This work is intended to match the target filesystem, FAT32, at a binary level and return the same data and error codes returned by mature FAT32 implementations, considered canonical. We provide a logical framework for reasoning about program behavior when filesystem calls are involved in terms of separation logic, and adapt it to simplify and automate the proof process in ACL2. By providing this framework, we encourage and facilitate greater adoption of software verification by application developers.

Keywords: Filesystems · Theorem Proving · Separation Logic.

1 Introduction

Filesystems have been of interest to the formal verification community for a while. This paper discusses our recent work on supporting code proofs for programs making use of a verified filesystem, using separation logic. Although alternative approaches towards code proofs exist, we find separation logic to be a natural choice for its intuitiveness as well as its easy applicability towards verifying code through term rewriting as implemented in general purpose theorem proving systems.

This work introduces AbsFAT, an axiomatization of a subset of the POSIX system calls offered by filesystems, specifically FAT32. We build on our earlier work [18] developing the FAT32 models HiFAT and LoFAT in the ACL2 general-purpose theorem prover; our axiomatic approach here complements the refinement approach the earlier work advocated. Although refinement remains necessary to keep tractable the problem of verifying the binary compatible model

$\{True\}$
r := mkdir(path); s := stat(path)
$\{(r \Rightarrow 0) \rightarrow (s \Rightarrow 0)\}$

Fig. 1: A simple conjecture involving mkdir and stat

LoFAT, axiomatization is needed in order to prove properties about external programs which are not (and should not be) developed with the details of our formalization in mind, except for the POSIX-like interface that LoFAT offers.

Fig. 1 is an example of a simple external program that calls mkdir, a system call for creating a directory at the given path, and then stat, a system call for checking the existence of a file at given path and obtaining pertinent metadata (which we overlook for the purposes of this proof). This example conjectures a Hoare triple with the precondition True and the postcondition $(r \Rightarrow 0) \rightarrow (s \Rightarrow 0)$. This conjecture says that stat succeeds when mkdir succeeds, and proving it requires precise specifications of these system calls both in terms of changes in filesystem state and in terms of return values and success/error conditions. AbsFAT provides such specifications, and ACL2 is able to prove this conjecture automatically through rewriting with AbsFAT. Programmers working at this level of abstraction rely on these return values and success/error conditions, around which they build their program logic; this makes the present work valuable.

Our principal contributions are

- the development of AbsFAT on the basis of abstract separation logic;
- the formalization of its connection to the existing models HiFAT and LoFAT, which involves the development of a number of system calls and showing that they are equivalent to the versions of those system calls in HiFAT and in turn in LoFAT;
- the development of a library of lemmas which facilitate a number of Hoare-style code proofs through rewriting, accompanied by examples of such proofs.

We detail these contributions in the rest of the paper, providing necessary background about program reasoning and FAT32 in Section 2, detailing our approach towards and implementation of abstract separation logic in Section 3, and evaluating our work in Section 4.

2 Background

2.1 ACL2

ACL2 [17] is an interactive theorem prover for first-order logic. Prior work on program verification with ACL2 has highlighted its library support for code proofs based on executable models [9] and its support for rewriting-based proofs of separation properties in the context of program verification [22] through the use of proof tactics (i.e. rewrite rules) which act upon a quantifier-free representation of separation predicates.

2.2 Program logic

Floyd-Hoare logic [11,7], which introduced Hoare triples

$\{precondition\}\ statement\ \{postcondition\}$

for simple and compound statements comprising a program, forms the basis of program verification. In recent years, the logic has been strengthened in many different program verification domains, and been successfully automated for static analysis both at the programming language level and at the machine code level. In this paper, we are interested in Floyd-Hoare logic as applied to linearly addressed memory for storage of fixed-size elements, such as bytes. Such memory models have been developed to model RAM, which usually provides byte- or word-addressing; we find the idea immediately applicable to filesystems, which operate on various kinds of linearly addressed storage media while abstracting away the details of those media to expose a directory tree interface to the user.

One such extension is separation logic [25,26], which was initially developed to address certain *scalability* challenges in reasoning about programs operating on a RAM-like memory model, i.e. the *flat heap* model. Heaps are mappings from locations (*heap cells*) to fixed-size data values; they can be composed using the *separating conjunction* operator $*$. The heap $P * Q$ is defined as the union of the mappings P and Q, along with the assertion that no mappings are common between P and Q which is implicit whenever $P * Q$ appears in a Hoare triple. Yang et al. [30] describe a bargain of sorts: in exchange for using only *tight specifications*, i.e. Hoare triples $\{P\}\,stmt\,\{Q\}$ in which the precondition P names all the locations modified by *stmt*, we get a *frame rule* allowing us to infer from this Hoare triple the new Hoare triple $\{P * R\}\ stmt\ \{Q * R\}$. This derivation is sound since the formula P * R contains the assertion that locations in P and R are disjoint from each other; and the new Hoare triple preserves tightness.

Structured Separation Logic (SSL) [29], an extension of this theory, replaces the flat heap with a *structured heap*. Here, a heap cell is allowed to be either a flat heap cell containing a fixed-size value, or a structured heap cell containing a rich value such as a linked list or a sublist thereof. The utility in reasoning about filesystems is immediately obvious, when we consider rich values to include the contents of directories.

In SSL, the notion of a structured heap goes together with the notion of an *abstract heap* which helps reason about accesses and updates to substructures of structured data. Like structured heaps, abstract heaps contain flat heap cells and structured heap cells; in addition, they contain *abstract heap cells*, pairs of values each consisting of a substructure of some structured data and the path where the substructure fits in the larger structure. When a substructure is placed in an abstract heap cell, at the same time the structure where it fits is changed to replace the substructure with a *body address* pointing to this heap cell. This process of creating an abstract heap cell is known as *abstract allocation*; it serves to create a new cell in the abstract heap and "separate" a substructure. The converse operation is *abstract deallocation*; it does away with an abstract heap cell by folding its contents back into the larger structure whence they came, in a manner similar to application of functions in lambda calculus. Abstract

allocation and abstract deallocation can both be carried out at will depending on the needs of the proof, since neither has any effect on the state of the system. Since Hoare judgments about sequences of commands often require abstract allocations and deallocations to be written out explicitly, we use Hoare triples of the form $\{P\}\, id\, \{Q\}$, to represent them, where id is a vacuous statement (i.e. a no-op).

The path stored in each heap cell is known as a *path promise*, ensuring that the substructure in the first element is identifiable as to its origin in the substructure. In the context of SSL applied to filesystems, this marks a difference in approach from prior work in specifying filesystems [10,20] which choose to index into the heap by file path, making the path of each file its heap address. While a variable may be deallocated and thus removed from the abstract heap, its path promise cannot change as long as it is in the abstract heap.

2.3 FAT32

FAT32 has been widely used, on personal computers where it was for several years the Windows operating system's default, and on removable media and embedded devices where it continues to be ubiquitous. Microsoft provides an authoritative specification [19] which specifies the on-disk layout of the filesystem in terms of file data, directory-level metadata and volume-level metadata. Two pertinent details follow.

- FAT32's data region is indexed by its file allocation table, which holds a linked list for each file called a *clusterchain*. Looking up a file's clusterchain in the data region yields its contents. Clusterchains are not shared between files; thus, there is no hard linking.
- Each directory in FAT32 is explicitly required to include, in addition to directory entries for regular files and subdirectories, an entry for itself (.) and its parent (..).

Our previous work [18] introduced LoFAT and HiFAT, two models at different levels of abstraction. LoFAT is a disk-image level model of FAT32, while HiFAT is a directory tree-based model of FAT32 which abstracts LoFAT. LoFAT includes the capability to read its state from a FAT32 disk image and write back its state to a disk image; efficient implementations of both of these make the model executable. The efficient execution of the model supports a suite of co-simulation tests for programs from the Coreutils and mtools sets of programs; in each of these tests, the canonical program is run with the same input as the ACL2 test program to check that the standard outputs of these two programs, if any, are identical and filesystem states, if changed, are identical as well. The system calls in LoFAT are proved to be refinements of the corresponding system calls in HiFAT, because the latter is an easier model to reason about. These refinement properties rest on transformations between the two models and proofs that these transformations are inverses of each other under appropriate equivalence relations.

$$\{(\mathsf{path} \Mapsto P/b/a) * (\alpha^P \mapsto b[C * \mathit{can\_create}(a)])\}$$
$$\mathsf{r} := \mathsf{mkdir}(\mathsf{path})$$
$$\{\mathsf{r} \Rightarrow 0 * \alpha^P \mapsto b[C + a[\varnothing]]\}$$

$$\{(\mathsf{path} \Mapsto P/b/a) * (\alpha^P \mapsto b[C + a : s])\}$$
$$\mathsf{r} := \mathsf{mkdir}(\mathsf{path})$$
$$\{\mathsf{r} \Rightarrow \mathsf{EEXIST}\}$$

$$\{(\mathsf{path} \Mapsto P/b/a) * (\alpha^P \mapsto b[C + a[\beta]])\}$$
$$\mathsf{r} := \mathsf{mkdir}(\mathsf{path})$$
$$\{\mathsf{r} \Rightarrow \mathsf{EEXIST}\}$$

$$\{(\mathsf{path} \Mapsto P/b/a) * (\alpha^P \mapsto b : s)\}$$
$$\mathsf{r} := \mathsf{mkdir}(\mathsf{path})$$
$$\{\mathsf{r} \Rightarrow \mathsf{ENOTDIR}\}$$

$$\{(\mathsf{path} \Mapsto P/a) * \mathsf{ENOENT}(P)\}$$
$$\mathsf{r} := \mathsf{mkdir}(\mathsf{path})$$
$$\{\mathsf{r} \Rightarrow \mathsf{ENOENT}\}$$

Fig. 2: Hoare triples describing the possible behaviors of mkdir

3 Abstract Separation Logic for Filesystem Verification

In AbsFAT, the model for reasoning about filesystem calls in programs, filesystem states are represented as collections of logically separate variables. Maintaining this separation through state changes from applications of system call specifications, abstract allocations, and abstract deallocations is a substantial part of how the proofs are resolved in the theorem prover. This formal development is structured around these principles:

- To profit from separation logic, properties of the filesystem state are represented as local properties where possible, in a way that allows them to be proved by rewriting without induction.
- Global properties, which pertain to the whole system and which have the potential to expand the state space, are packaged into disabled predicates in order to keep the state space manageable (for the prover) and make them more comprehensible (to the user.) This, in turn, leads to the creation of general and reusable lemmas to resolve subgoals involving these predicates which arise from file operation specifications.
- Also in the interest of reducing state space and helping user comprehension, file operation specifications are kept compact.
- Wherever new global properties are required specifically for FAT32, they are distinguished where possible to simplify possible reuse of the proof infrastructure for a future filesystem formalization.
- A completion semantics is used, in which the effect of a system call is deterministically evaluated regardless of whether it returns an error code.

3.1 System calls and support for filesystem clients

It is illustrative to more closely examine Fig. 1 and its proof, which in turn requires us to consider the specification of mkdir itself. This system call has one success condition and several error conditions, as shown in Fig. 2, where $\mathsf{path} \Mapsto P/b/a$ means that path matches a directory P with subdirectory b, which has a component a; $b[C * \mathit{can\_create}(a)]$ describes a directory b in which

a can be created; $b[C + a : s]$ describes a directory b where regular file a already exists with contents s, $b[C + a[\beta]]$ describes a directory b where directory a exists with contents β, and $b : s$ describes a regular file b with contents s. In all but the first of these five Hoare triples, mkdir exits returning an error code, as standardized by POSIX which defines the meanings for each one of the errors ENOENT, ENOTDIR, ... and their applicability to a given system call.[1]

The AbsFAT axiomatization works on frames, collections of logically separate abstract variables, and supports claims such as the following which is likely to arise in the context of any proof involving the system call mkdir:

> When a given filesystem state corresponds to a given well-formed separation logic frame, the state after a successful call to mkdir(path) corresponds to a new well-formed frame with all abstract variables unrelated to dirname(path) unchanged, and with a new variable α representing the current state of dirname(path) containing a new empty directory named basename(path).

This claim, expressed in terms of the Linux utility functions basename [13] and dirname [14], is an example of the benefits of local reasoning. It expresses the new state after an operation as a change to the old state that clearly leaves the remainder of the filesystem unchanged and simplifies reasoning about the next operation on the filesystem in both cases, when this next operation affects the directory tree under dirname(path) and when it does not. Since our system makes filesystem operations deterministic, including at the AbsFAT level, we are able to make the stronger claim:

> When a given filesystem state corresponds to a given well-formed separation logic frame, the state after a call to mkdir(path), successful or not, corresponds to a new well-formed frame with all abstract variables unchanged which are logically separate from the contents of basename(path). Further, when dirname(path) exists as a directory and no file exists at path, the frame will have a new variable α representing the current state of dirname(path) containing a new empty directory named basename(path).

The proof of the Fig. 1 conjecture splits into a number of cases, as expected from our understanding of the mkdir specification; Fig. 3 shows these case splits.

[1] We use the values ENOENT, ENOTDIR as the return values of system calls, but POSIX enumerates these as values assigned to the global variable errno, which is different from the function's actual return value. Our implementation of each system call must return both these values, in order to capture the entirety of the effect of the system call in ACL2, a functional language lacking global program state. For brevity in this paper, however, we find it convenient to refer to a single return value which is zero in the case of success, and one of the errno values enumerated by POSIX [15] in the case of failure. This is no less informative than providing the return value and the errno value separately, since none of our system calls sets errno to 0 upon success.

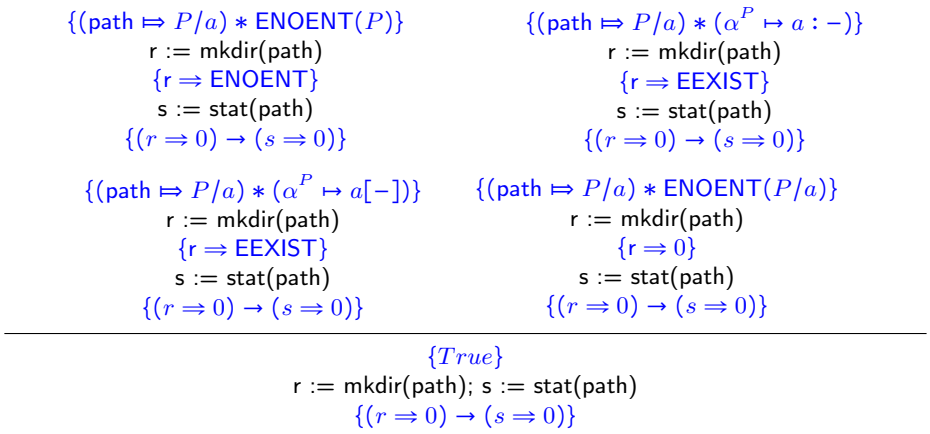

Fig. 3: Proof tree for the conjecture in Fig. 1

While all but one of the cases are vacuous, since the postcondition implication $\{(r \Rightarrow 0) \rightarrow (s \Rightarrow 0)\}$ is vacuously true whenever r is nonzero, the cases still need to be explored for soundness of the proof. The justification for developing the theorem proving infrastructure becomes apparent simply from exploring the number of cases arising in this proof for a trivial sequence of two operations, which is not conducive to hand proofs.

As one might expect, a complementary conjecture that mkdir leaves the filesystem state unchanged in an error condition, i.e. when $\neg(r \Rightarrow 0)$, is also automatically provable with the help of AbsFAT.

3.2 Rewriting

AbsFAT's abstract separation-logic based formulation of file operations has a logically straightforward representation in the theorem proving environment, which connects neatly with the existing HiFAT directory-tree representation. Thus, a valid filesystem instance can be represented as a *frame*, an association list mapping variables to unrooted trees in which the root is distinguished from the other variables.

The usability of a proof technique depends, to a significant extent, on the degree to which it can be automated by the prover, freeing the user from the manual instantiation of lemmas and other laborious aspects of theorem proving. In ACL2, this is naturally the task of the rewriter, which efficiently completes proofs even when they involve large terms of the kind that show up in code proofs. Accordingly, we look for a method to phrase the separation properties in a way that is palatable to the rewriter, after the fashion of Myreen and Kaufmann's work on code proofs [22] demonstrating a logical workaround for the difficulties of separation logic proofs in the presence of existential quantifiers. The recursive predicate *separate* served to take the place of the conventional

$$\mathsf{old} \Rrightarrow P/a * \mathsf{new} \Rrightarrow P'/b * \alpha^{P} \mapsto a[C] * \beta^{P'} \mapsto b[C']$$
$$\mathsf{rename\ old\ new}$$
$$\alpha^{P} \mapsto \phi * \beta^{P'} \mapsto b[C' + a[C]]$$

Fig. 4: An axiom that means different things in the HiFAT and LoFAT contexts

separating conjunction operator, which by definition introduces the unwanted existential quantifiers. In proofs, the *separate* predicate under the action of certain rewriting tactics was shown to create those case splits which were necessary to resolve the subgoals and complete the proof. Our formulation of *separate* is similar, and helps us avoid existential quantifiers.

In the general setting where zero or more abstract variables exist at body addresses within the frame frame, and each corresponds to an abstract heap cell in the heap, we can recursively define the predicate separate, in Haskell notation:

$$\mathsf{separate}\ [\,] = \mathit{True}$$
$$\mathsf{separate}\ ((\alpha, \mathit{entry}_\alpha) : \mathit{frame}) = (\mathsf{disjoint}\ \mathit{frame}\ \mathit{entry}_\alpha) \wedge (\mathsf{separate}\ \mathit{frame})$$

This predicate has an analogous effect to Myreen's predicate in the flat memory model, creating case splits which help resolve the subgoals of the proof. This holds true even though the filesystem model is more complex with the abstract variables needing to be maintained in a well-formed state in the sense of collapsibility. Collapsibility, which needs to be maintained as an invariant through various file operations, is necessarily a global property; separate facilitates reasoning about it locally and generally minimizes the need for non-local lemmas.

A choice, here, is whether separate and abstract allocation/deallocation should be implemented at the LoFAT level, or the HiFAT level. At a first glance, it seems useful to reason about clusterchains in LoFAT and declare them disjoint through the separate predicate; however, these disjointness properties are adequately addressed by the definition of lofat-to-hifat, a function which transforms a LoFAT instance to a HiFAT instance and returns an error for ill-formed LoFAT instances, including those with any overlap between clusterchains. The choice is ultimately determined by a FAT32-specific consideration: a subdirectory is not completely independent of its parent directory, because it needs to have an explicit .. directory entry pointing to its parent. Thus, a naïve implementation over LoFAT would cause inconsistencies; one such inconsistency arises in one of the axioms for the system call rename (Fig. 4), which addresses the moving of a subdirectory. This axiom implies that the contents of the subdirectory b are unchanged, which is untrue for LoFAT since the .. directory entry is updated from P to P'/b. This axiom holds true for HiFAT though, which has no . or .. entries since it's a directory tree model and doesn't need them. Thus, we opt to implement this logic at the HiFAT level, avoiding inconsistencies such as the above in AbsFAT and generally benefitting from HiFAT's ease of use for reasoning.

AbsFAT deals with filesystems, instrumented with body addresses and accompanied by heaps which bind each abstract variable in some body address to

a *partial directory*, i.e. a fragment of a directory, with none, some, or all of that directory's contents. AbsFAT refers to such entities as frames, assisting the intuition that a frame contains some immediately useful information and some less immediately useful information, which the frame rule of separation logic helps in managing.

The well-formedness of such a frame is determined by Gardner et. al's [8] definition of the one-step collapse relation $\downarrow$, which describes a single context application folding an abstract heap cell into the incomplete directory tree holding the corresponding body address, and its transitive closure $\downarrow^*$. A frame is well-formed if it is related by $\downarrow^*$ to some frame consisting of only the root variable, τ, bound to a *complete directory tree* which contains no body addresses. This definition is, self-evidently, existentially quantified over all frames that the given frame could be related to, and is therefore a difficult definition to work with in ACL2 which has limited support for quantified predicates.

It is intuitive, however, and a fact we prove in ACL2, that if a given frame is related via $\downarrow^*$ to two complete directory trees then these two trees must have the same directory structure and contain the same files modulo rearrangement of files within subdirectories. Thus, AbsFAT opts for an alternative formulation of transitive collapse, iterating over all abstract heap cells until all are folded into the root variable (or until no variable can be folded, which shows the frame to be ill-formed). This iterative formulation simplifies the task of maintaining well-formedness as an invariant across file operations from a reasoning perspective, and also makes the invariant intuitive: the property being preserved is the ability of the frame to be properly collapsed.

An instrumented filesystem consists of a root directory and an association list mapping each numbered variable to:

- a partial directory, possibly containing body addresses itself;
- a path promise; and
- a source pointer to an abstract variable (possibly the root itself) containing a body address for the given variable.

The well-formedness of such an instrumented filesystem becomes the proposition of whether it can be collapsed without errors; i.e. whether every abstract cell corresponds to a body address in some variable, matching its path promise. As a basic property of our formulation, we prove that a successful collapse as described above yields a HiFAT instance with no duplicate filenames in any directory, under the hypotheses of separation between the different variables in the separate sense and the absence of "dangling" body addresses which lack corresponding abstract cells.

The α values are chosen to be natural numbers to simplify the bookkeeping involved in the implementation; and disjointness requires all of them to be distinct from each other. Each $entry_\alpha$ value is a partial directory. These entries

match Gardner et al.'s definition of an unrooted tree[2], given by the grammar:

$$ud \ ::= \ \phi \ \mid \ a : s \ \mid \ a[ud] \ \mid \ ud + ud \ \mid \ x$$

In this grammar, ϕ is the empty tree, $a : s$ is a regular file with name a and contents s, $a[ud]$ is a directory named a and containing unrooted tree ud, $ud+ud$ is composition, and x is a variable. This definition allows for multiple variables in a directory, as does our code, but in practice we find it useful while reasoning about file operations to avoid having more than one variable in a directory. When this property holds, it becomes possible to check the existence of a file at a given path by examining the contents of at most one abstract variable.

It remains to address one point: how are abstract allocations to be done in a proof? In the proof tree in Fig. 3 for instance, the $r \Rightarrow 0$ case assumes the existence of variable α^P with the appropriate path promise and the appropriate contents; in a proof search the variable needs to be allocated procedurally. A naïve answer would be to collapse the entire frame into a single root variable and then allocate a new variable with the desired path promise, creating the corresponding body address in the root. This is undesirable, limiting if not entirely curtailing the local reasoning we hope for by bringing all variables together.

AbsFAT solves this question by implementing a transformation on the frame of abstract variables, called *partial collapse*. For a given path, this operation iteratively deallocates every abstract variable with a path promise prefixed by this path, leaving the frame in a state where the contents of the directory at that path are to be found, without holes, in one variable. All other variables are left as is. This is essential to the specification of almost every one of the system calls we consider, since they require some directory to be brought into a variable for examination, the way α^P in the mkdir(path) specification brings in the contents of basename(path). For instance:

- In the specification of mkdir(path), partially collapsing the frame on dirname(path) allows the contents of the parent directory to be seen, which determines whether the system call will succeed (when no file exists at path) or fail (when a regular or directory file does exist at path.)
- Similarly, in the specification of stat(path), partially collapsing dirname(path) reveals the contents of the parent directory which determines whether the system call will succeed (when a regular or directory file does exist at path) or fail.

Using partial collapse necessitates a somewhat complex proof to show that a partially collapsed frame collapses to the same filesystem as it would have before the partial collapse, modulo rearrangement of files within directories. The payoff for this proof is in the form of clean specifications of system calls, deterministically evaluating the state of the filesystem just like the system call specifications in HiFAT and LoFAT. The specifications capture the read-over-write properties of the implementations of HiFAT and LoFAT, while avoiding

[2] We replace inodes in the original grammar with strings. In FAT32, with no hardlinks, it is easier to just represent the contents of regular files as strings.

```
{
r := mkdir("/tmp/docs")
s := mkdir("/tmp/docs/pdf-docs")
r ⇒ 0 ∗ s ⇒ − ∗ t ⇒ −
t := stat("/tmp/docs")
r ⇒ 0 ∗ t ⇒ 0
}
```

Fig. 5: Reasoning about longer sequences of system calls

one of the key problems, namely, the general difficulty of getting read-over-write theorems to apply because of various simplifications in the course of a rewriting proof that keep the left-hand side of the rewrite rule from being unifiable with the target.

This leads to another example (Fig. 5), which is worked out in ACL2 along the same lines as demonstrated here. This exemplifies the kinds of properties we need to prove across sequences of operations, when sometimes we need to logically disregard the effect of one system call such as the second mkdir in this case.

This example also shows one interesting approach to verifying filesystem properties, in which the precondition for one system call to do the desired thing may be excessively verbose to write out in full, but can be more concisely expressed in terms of the return value of a previous system call.

3.3 LoFAT-HiFAT correspondence

Many of the system calls in LoFAT are implemented through three primitive operations: lofat-find-file, lofat-remove-file, and lofat-place-file. For instance, the system call mkdir is implemented as a call of lofat-place-file with an empty directory as an argument. Thus, the proofs of specifications for these system calls reduce to proofs about these three primitives.

More precisely, we prove the HiFAT versions of these system calls, respectively hifat-find-file, hifat-remove-file, and hifat-place-file, to have properties as required by the specifications for the system calls. These turn out to be read-over-write properties, adapted as the case may be for "writes" to the filesystem which are file creations, file updates and file deletions. In a general form, these properties state:

1. A read from the filesystem following a write at the same location yields that which was written.
2. A read from the filesystem following a write at a different place yields the same result as a read before the write.

Then, by proving the refinement relationship for each of these three primitives in LoFAT, we are able to much more easily prove that LoFAT meets specifications. LoFAT is a linearly addressed data structure, since it emulates the layout

of an actual FAT32 disk image; thus, any proofs regarding operations on LoFAT by necessity involve separation properties between different files. The executable function lofat-to-hifat, which transforms a LoFAT instance to a HiFAT instance, checks whether the clusterchains across all the different files in a filesystem are disjoint from each other. This disjointness is a necessary property for a well-formed FAT32 disk image, and therefore a necessary property of a well-formed LoFAT instance. However, such checks on the disjointness of sets of clusterchains can get expensive in a large filesystem instance, which is why we deem it inadvisable to continue with the earlier implementation of LoFAT which effects one or more transformations between HiFAT and LoFAT for each system call. Therefore, we develop more efficient implementations of the LoFAT primitives lofat-find-file and lofat-remove-file; verify that each refines the respective HiFAT primitive; and adapt the system calls in LoFAT to use the new primitives.

We provide specifications for the system calls in LoFAT, through which programs making use of the filesystem can be reasoned about. LoFAT functions at one level of remove from the disk image, since it is a single-threaded object; however, it replicates the structure of the disk in memory and each system call it offers is an analogue of the system call of the same name, adapted for the use of programs written in the applicative ACL2 language. Some of the system calls implemented in LoFAT involve file descriptors; as a result file descriptor tables and file tables are implemented as part of LoFAT. This means that, viewing the disk image as a linearly addressable entity and disregarding details about caching, each execution of a program using FAT32's POSIX interface is mirrored by an execution of an ACL2 program using LoFAT with the same program logic but making use of LoFAT's system calls; and any proofs carried out on the latter also apply to the former. This correspondence underlies the focus on binary compatibility and efficient execution in the present work.

A significant part of the implementation has been the reworking of the LoFAT primitives lofat-find-file and lofat-remove-file. The latter now guarantees that files not in the path of a file operation will remain unchanged in their placement in the data region, in contrast to the earlier implementation which transformed to HiFAT, performed the removal, and transformed back to HiFAT, effectively reallocating space in the data region for every file in the filesystem. For proving that these functions refine their HiFAT equivalents, respectively hifat-place-file and hifat-remove-file, we opted to simplify the specifications of these functions to make them *no-change losers*[3]: functions which operate on a data structure making sure to return it unchanged if returning an error code indicating failure. This yielded rewrite rules with fewer hypotheses, which made it simpler to prove the lemmas necessary to support our filesystem call specifications.

4 Evaluation

In evaluating this system, we account for the ease of writing and verifying programs using LoFAT. To demonstrate that realistic programs can indeed be writ-

[3] This is a term from ACL2 lore, and often used in the ACL2 source code.

cp, ls, mkdir, mv, rm, rmdir, stat, truncate, wc

Fig. 6: Coreutils programs in the co-simulation test suite

ten, we develop a test suite of programs which emulate existing programs from the Coreutils. Building upon the tests already developed our earlier work on HiFAT and LoFAT, we add support for the system calls readdir, opendir, closedir and truncate; following this, we are able to write a program emulating the Coreutils program truncate. Each of the programs in our test suite is compared in terms of output and filesystem state to the program it emulated in one or more `co-simulation tests`; programs which write to standard output are tested for character-by-character correspondence to the canonical programs, and programs which change the state of the filesystem are compared for file-by-file correspondence to the state after running the canonical program with the same arguments Fig. 6 summarizes the programs which are tested in the co-simulation test suite.

The proof development for supporting the filesystem proofs demonstrated in this paper is necessarily complex, since AbsFAT needs to abstract HiFAT and in turn LoFAT without loss of information. Examining the code supporting the basic frame operations collapse and partial collapse, the different system calls, the proofs that these system calls abstract the corresponding system calls in HiFAT and the proofs of the examples developed in this paper, we count 33,106 lines of code across 79 function definitions and 1143 lemmas. It remains of interest to check if ACL2's features for automatically generating lemmas [1], already used in LoFAT, could be brought to bear on some of these.

5 Related work

Much of the prior work has developed new verified filesystems, making crash consistency a priority. Yggdrasil [28], a verification toolkit, employing SMT solving in Z3 [21] but eschewing explicit separation reasoning, delivers the Yxv6 filesystem which has verified crash consistency properties. COGENT [2], a verifying compiler, provides a domain specific language and generates executable filesystem code from a specifications in that language, accompanied by proofs of correctness in Isabelle/HOL [23]. FSCQ [6], which introduces Crash Hoare Logic, is the first in a family of related filesystems verified in Coq [3] which also includes DFSCQ [5], a filesystem providing and adhering to a formal specification for the fdatasync system call, and SFSCQ [12], a filesystem with verified security properties. While separation logic is part of the development of Crash Hoare Logic in FSCQ, that effort does not include reasoning at the file operation level.

The work of Gardner et al. on specifying preconditions and postconditions of POSIX filesystem calls [8] has been extended into a theory of concurrent

operation of filesystems [24]. This work, however, has not attempted to deliver an executable filesystem upon which programs could be run.

Two tangentially related systems with similar aims to AbsFAT are the work of Koh et al. [16] towards verifying the operating system components involved in a networked server, and the work of Chajed et al. [4] on Argosy, a verified system for stacking storage layers. While these systems are executable, they do not offer the sort of support for constructing and simplifying proofs about filesystem clients that AbsFAT does.

6 Conclusion

This paper describes AbsFAT, a separation logic model which formally describes the different system calls in LoFAT, a faithful executable model of FAT32. AbsFAT's formal descriptions of these system calls support proofs of correctness for programs which interact with the filesystem. The examples in the paper demonstrate the kinds of Hoare triples that arise in this verification context and how ACL2 completes these proofs by rewriting guided by AbsFAT's separation logic formulation. This application of abstract separation logic to an executable filesystem model is novel and shows several general principles which are applicable to the verification of filesystems other than FAT32 and the programs which make use of them.

An informal survey of filesystem-related bug reports yielded the interesting examples of Shareaza, a file sharing application [27]. A bug in Shareaza caused an infinite loop when files larger than 4 GB were to be saved to a FAT32-formatted partition. Such an operation is disallowed by the published FAT32 specification in the interests of keeping clusterchains to a manageable length and the error code EFBIG is designated for filesystem implementations to indicate an error of this nature. This sort of bug, which could be mitigated by attention to return values and error codes, is our motivation for building the precise filesystem models this paper discusses.

Availability

The proofs and the cosimulation tests discussed in the paper can be found in the books/projects/filesystem subdirectory of the ACL2 distribution, which is available at `https://github.com/acl2/acl2/`. Instructions for certifying the books in ACL2 are provided in README.md in that subdirectory.

References

1. ACL2 Community: ACL2 documentation for MBE, see URL `http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2___MAKE-EVENT`

2. Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., et al.: Cogent: Verifying high-assurance file system implementations. ACM SIGPLAN Notices **51**(4), 175–188 (2016)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer Berlin Heidelberg (2004)
4. Chajed, T., Tassarotti, J., Kaashoek, M.F., Zeldovich, N.: Argosy: Verifying layered storage systems with recovery refinement. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 1054–1068. ACM (2019)
5. Chen, H., Chajed, T., Konradi, A., Wang, S., İleri, A., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Verifying a high-performance crash-safe file system using a tree specification. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP). pp. 270–286. ACM (2017)
6. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using Crash Hoare Logic for certifying the FSCQ file system. In: USENIX Annual Technical Conference (USENIX ATC 16). USENIX Association (Jun 2016)
7. Floyd, R.W.: Assigning meanings to programs. Proceedings of the American Mathematical Society Symposia on Applied Mathematics **6**, 19–31 (1967)
8. Gardner, P., Ntzik, G., Wright, A.: Local reasoning for the posix file system. In: Programming Languages and Systems. pp. 169–188. Springer Berlin Heidelberg (2014)
9. Goel, S.: Formal verification of application and system programs based on a validated x86 ISA model. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin (2016)
10. Hesselink, W.H., Lali, M.: Formalizing a hierarchical file system. Electronic Notes in Theoretical Computer Science **259**, 67 – 85 (2009), proceedings of the 14th BCS-FACS Refinement Workshop (REFINE)
11. Hoare, C.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
12. Ileri, A., Chajed, T., Chlipala, A., Kaashoek, F., Zeldovich, N.: Proving confidentiality in a file system using DiskSec. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 323–338. USENIX Association (Oct 2018)
13. Kerrisk, M.: basename (3)-Linux manual page, accessed: 01 Aug 2020
14. Kerrisk, M.: dirname (3)-Linux manual page, accessed: 01 Aug 2020
15. Kerrisk, M.: errno (3)-Linux manual page, accessed: 01 Oct 2019
16. Koh, N., Li, Y., Li, Y., Xia, L.y., Beringer, L., Honoré, W., Mansky, W., Pierce, B.C., Zdancewic, S.: From C to Interaction Trees: Specifying, verifying, and testing a networked server. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP). pp. 234–248. ACM (2019)
17. Matt Kaufmann, Panagiotis Manolios, and J S. Moore: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (Jun 2000)
18. Mehta, M.P., Cook, W.R.: Binary-Compatible Verification of Filesystems with ACL2. In: 10th International Conference on Interactive Theorem Proving (ITP). Leibniz International Proceedings in Informatics (LIPIcs), vol. 141, pp. 25:1–25:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2019)
19. Microsoft: Microsoft extensible firmware initiative FAT32 file system specification (Dec 2000), `https://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc`

20. Morgan, C., Sufrin, B.: Specification of the unix filing system. IEEE Transactions on Software Engineering **SE-10**(2), 128–142 (March 1984)
21. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg (2008)
22. Myreen, M.O.: Separation logic adapted for proofs by rewriting. In: Interactive Theorem Proving. pp. 485–489. Springer Berlin Heidelberg (2010)
23. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, vol. 2283. Springer Berlin Heidelberg (2002)
24. Ntzik, G., da Rocha Pinto, P., Sutherland, J., Gardner, P.: A Concurrent Specification of POSIX File Systems. In: 32nd European Conference on Object-Oriented Programming (ECOOP). Leibniz International Proceedings in Informatics (LIPIcs), vol. 109, pp. 4:1–4:28. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2018)
25. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Proceedings of the 15th International Workshop on Computer Science Logic (CSL). pp. 1–19. Springer-Verlag (2001)
26. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 55–74. IEEE Computer Society (2002)
27. Bug: file-system file-size limit handling (2011), `http://shareaza.sourceforge.net/phpbb/viewtopic.php?f=7&t=1118`
28. Sigurbjarnarson, H., Bornholt, J., Torlak, E., Wang, X.: Push-button verification of file systems via crash refinement. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 1–16. USENIX Association (Nov 2016)
29. Wright, A.: Structural separation logic. Ph.D. thesis, Imperial College London (2013)
30. Yang, H., O'Hearn, P.: A semantic basis for local reasoning. In: International Conference on Foundations of Software Science and Computation Structures. pp. 402–416. Springer (2002)