Verification of a Virtual Filesystem Switch *

Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg, Germany
{ernst,schellhorn,haneberg,joerg.pfaehler,reif}
@informatik.uni-augsburg.de

Abstract. This work presents part of our verification effort to construct a correct file system for Flash memory. As a blueprint we use UBIFS, which is part of Linux. As all file systems in Linux, UBIFS implements the Virtual Filesystem Switch (VFS) interface. VFS in turn implements top-level POSIX operations. This paper bridges the gap between an abstract specification of POSIX and a realistic model of VFS by ASM refinement. The models and proofs are mechanized in the interactive theorem prover KIV. Algebraic directory trees are mapped to the pointer structures of VFS using Separation Logic. We consider hard-links, file handles and the partitioning of file content into pages.

Keywords: Flash File System, Verification, Refinement, POSIX, VFS, Separation Logic, KIV

1 Introduction

The popularity of Flash memory as a storage technology has been increasing constantly over the last years. It offers a couple of advantages compared to magnetic storage: It is less susceptible to mechanical shock, consumes less energy and read access is much faster. However, it does not support overwriting data in-place. This limitation leads to significant complexity in the software accessing Flash memory. One solution is a Flash translation layer (FTL) built into the hardware, which emulates the behavior of magnetic storage. Embedded systems, however, often contain "raw Flash", which requires specific Flash file systems (FFS for short) that deal with the memory's write characteristics. A state-of-the-art example is *UBIFS* [15], which is part of the Linux kernel.

The use of Flash memory in safety-critical applications leads to high costs of failures and correspondingly to a demand for high reliability of the FFS implementation. As an example, an error in the software access to the Flash store of the Mars Exploration Rover "Spirit" nearly ruined the mission [22]. In response, Joshi and Holzmann [16] from the NASA JPL proposed in 2007 the verification of an FFS as a pilot project of Hoare's Verification Grand Challenge [14].

* The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-642-54108-7_13

We are developing such a verified FFS as an implementation of the POSIX file system interface [29], using UBIFS as a blueprint. The project is structured into layers, as (partially) visualized in Fig. 1. These correspond to the various logical parts of the file system, and to different levels of abstraction.

The top level is an abstract formal model of the file system interface as defined by the POSIX standard. It serves as the specification of the functional requirements, i.e., what it means to create/remove a file/directory and how the contents of files are accessed. The POSIX interface addresses files and directories by paths and views files as a linear sequence of bytes. File system objects are structured hierarchically as a tree. Directories correspond to the inner nodes of the tree, whereas files are found at the leaf nodes.

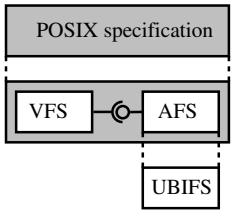

Fig. 1. Upper layers

The first contribution of this work is a formal POSIX model that supports all essential file system operations.

Such high-level concepts are mapped to an efficient pointer-based data representation in the file system. In Linux as well as in our approach this mapping is realized by a *Virtual Filesystem Switch* (VFS). The analogous component in Windows is named Installable File System (IFS). This layer implements generic operations that are common to all file systems, e.g., mapping of file content to a sparse array of pages, permission checks and management of open file handles. VFS delegates lower-level operations to concrete file systems, such as UBIFS.

We have recently published a formal VFS model [6]. It contains an abstract sub-specification AFS of the expected behavior of concrete file systems. The idea is that AFS can be replaced by a concrete implementation as long as the latter behaves as specified by AFS. The VFS model calls AFS through an internal interface, visualized by the symbol ⊸ in Fig. 1. A benefit of this approach is that AFS is independent of the characteristics of Flash memory and may serve as specification for traditional file systems as well, e.g., Ext2-4, ReiserFS or FAT.

Functional correctness is established by nested *refinements* (visualized as dashed lines in Fig. 1). For instance, a proof of the topmost refinement implies that the VFS model realizes the POSIX specification, and in particular that input-/output behavior is preserved.

We describe such a proof in this paper, which is the second contribution.

This refinement is conceptually challenging because of subtle requirements of the POSIX standard, and technically challenging because of the pointer structures and partitioning of file content into pages found in VFS. Models and proofs are mechanized in the interactive theorem prover KIV [23] and can be found on our website [7]. We also provide executable simulations (written manually in Scala) that integrate into Linux via FUSE [28]. As a consequence of the verification we can focus on the Flash File system's internals in the future, namely to refine AFS without further considering VFS. Formally, we refine AFS to UBIFS (in several steps), which then automatically guarantees correctness of VFS+UBIFS with respect to POSIX.

The text continues with a description of the approach in Sec. 2. Sections 3 and 4 describe the POSIX and VFS+AFS models; Sec. 5 formalizes the abstraction from VFS data structures to POSIX, Sec. 6 describes the proofs. We compare to related work in each of these three sections individually. Finally, Sec. 7 draws conclusions and points out ongoing and future work.

2 Scope and Approach

The purpose of this section is to give a high-level description of the approach. We consider the following structural POSIX system-level operations: `create`, `mkdir`, `rmdir`, `link`, `unlink`, and `rename`. File content can be accessed by the operations `open`, `close`, `read`, `write`, and `truncate`. Finally, directory listings and (abstract) metadata can be accessed by `readdir`, `readmeta` (= `stat`), and `writemeta` (subsuming `chmod`/`chown` etc).

These operations and the data types occurring in parameters constitute the signature of the system interface. Functionality is realized abstractly in the POSIX model and "concretely" by the VFS, which delegates low-level modifications to some concrete file system abstracted by AFS. POSIX and VFS share a common signature but have their own representation of the file system's state. By convention, we prefix operations with `posix_` resp. `vfs_`/`afs_` to distinguish between the different layers.

2.1 Formalism

Our specification language is based on *Abstract State Machines* [3] (ASMs). We use algebraic specifications to axiomatize data types, and a weakest-precondition calculus to verify properties.

We frequently use freely generated data types. For example, paths are defined by two constructors: the constant ϵ denoting the empty path, and an infix operator `/` that adds a leading component.[1] The corresponding selectors `first` and `rest` retrieve the constructor's arguments.

$$\textbf{data}\ \mathit{Path} = \epsilon \mid \_\,\texttt{/}\,\_(\texttt{first} : \mathit{String},\ \texttt{rest} : \mathit{Path})$$

We overload the symbol `/` to add a trailing path component p/s resp. to concatenate two paths p/p'.

Besides free types, we use partial functions types $\tau_1 \nrightarrow \tau_2$ in this work. All partial functions in this paper have a finite domain by construction as a non-free data type from the function with empty domain $\emptyset$ and function update $\_[\_ \mapsto \_]$. Partial function application $f[a]$ uses square brackets. Removing a from the domain of f is denoted by $f - a$. We use the abbreviation $a \in f$ for $a \in \texttt{dom}(f)$.

[1] Paths are actually defined as an instance of algebraic lists $\mathit{Path} := \mathit{List}\langle\mathit{String}\rangle$ plus some renaming. This paper deviates from the KIV specifications in minor details to aid readability. Such differences are noted on the web presentation.

ASMs maintain a state as a vector of logical variables that store algebraically defined data structures. The language features programming constructs such as parallel (function) assignments (where $f[a] := b$ abbreviates $f := f[a \mapsto b]$), conditionals, loops, recursive procedures, and also nondeterministic choice. ASMs are executable, provided that the nondeterminism is resolved somehow and the algebraic operations on data types are executable.

An ASM $M = ((\mathtt{OP}_i)_{i \in I}, \mathit{State}, \mathtt{INIT})$ consists of operations $\mathtt{OP}_i(\mathit{in}; \mathit{st}, \mathit{out})$ that take an input in and the current state $\mathit{st} : \mathit{State}$, and produce an output out and a modified state st'. The semicolon in the parameter list separates input parameters from reference parameters: assignments to the latter inside an operation are visible to the caller. Predicate $\mathtt{INIT} \subseteq \mathit{State}$ specifies a set of initial states. A run of an ASM starts in an initial state and repeatedly executes operations.

A "concrete" machine $C = ((\mathtt{COP}_i)_{i \in I}, \mathit{CState}, \mathtt{CINIT})$ *refines* an "abstract" machine $A = ((\mathtt{AOP}_i)_{i \in I}, \mathit{AState}, \mathtt{AINIT})$ if for each run of C there is a matching run of A with the same inputs and outputs. Refinement can be proven by forward simulation with a simulation relation $\mathtt{R} \subseteq \mathit{AState} \times \mathit{CState}$.

The calculus is based on sequents $\Gamma \vdash \Delta \equiv \forall \underline{x}. \bigwedge \Gamma \to \bigvee \Delta$ for a list of assumptions Γ, potential conclusions Δ and free variables of the sequent $\underline{x}$. We prove properties about ASM operations using the weakest precondition calculus implemented by KIV. It offers three modalities: the weakest precondition $\langle\!|p|\!\rangle\varphi$ of p with respect to φ (total correctness, all runs of p starting in the current state terminate in a state satisfying φ); the weakest liberal precondition $[p]\varphi$ (partial correctness); and $\langle p\rangle\varphi \equiv \neg[p]\neg\varphi$ that asserts the existence of some terminating run of p with a final state satisfying φ. For deterministic programs, $\langle\!|\_|\!\rangle\_$ and $\langle\_\rangle\_$ are equivalent. The calculus symbolically executes programs in modalities, reducing goals to predicate logic formulas.

The logic can express relationships between multiple programs, such as program inclusion or equivalence. In particular, proof obligations for data refinement [13] (as an instance of ASM refinement) can be formalized. Concretely, in this work we prove

initialization: (1)

$$\mathtt{CINIT}(cs) \vdash \exists as.\ \mathtt{AINIT}(as) \wedge \mathtt{R}(as, cs)$$

correctness: (2)

$$\mathtt{R}(as, cs) \vdash \langle\!|\mathtt{COP}_i(in; cs, out_1)|\!\rangle\langle \mathtt{AOP}_i(in; as, out_2)\rangle\,(\mathtt{R}(as, cs) \wedge out_1 = out_2)$$

for $A =$ POSIX and $C =$ VFS+AFS. These assertions establish a forward simulation from commuting 1:1 diagrams. Intuitively, "correctness" asserts that for each run of the concrete operation there is a matching abstract run with the same output, i.e., that the behavior of the concrete machine is covered by the specification/abstract machine. The predicate R relates a concrete state $cs : \mathit{CState}$ to an abstract state $as : \mathit{AState}$. It is composed of the invariants of the two machines and an abstraction relation

$$\mathtt{R}(as, cs) \;\leftrightarrow\; \mathtt{CINV}(cs) \wedge \mathtt{AINV}(as) \wedge \mathtt{ABS}(as, cs)$$

Background about ASM refinement and its relation to other refinement approaches can be found in [2,25].

2.2 Separation Logic

Separation Logic [24] is a logic designed to reason about pointer structures and destructive updates. It is particularly well-suited for structures with limited aliasing, such as the representation of the directory tree in VFS/AFS (Sec. 4).

Formulas in the logic are assertions $\varphi : \mathit{Heap} \to \mathbb{B}$ about the shape of heaps, which are mappings from locations to values, $\mathit{Heap} := (\mathit{Loc} \rightharpoonup \mathit{Val})$. Heap assertions are built from the constant $\texttt{emp} = (\lambda h.h = \emptyset)$, the maplet $l \mapsto v$ describing singleton heaps, and the separating conjunction $\varphi_1 * \varphi_2$ that asserts that the heap can be split into two disjoint parts satisfying φ_1 resp. φ_2.

Ordinary formulas, connectives and quantifiers are lifted over heaps, so that they can be used in separation logic assertions.

We have formalized separation logic as a straight-forward shallow embedding into higher-order logic, similar to [21,30]. For this work, we instantiate the sorts *Loc* and *Val* to the pointer structures used in VFS.

In our approach, the heap h is explicitly given as an ordinary program variable. This has the consequence that the frame rule for heap-modular reasoning is not generally valid. A counterexample is the non-local assignment $h := \emptyset$. Interestingly, this does not pose a problem in practice, as one can generalize contracts by a fresh placeholder variable f for the context, i.e., proving the frame rule for a particular contract for program p as $(\varphi * f)(h) \vdash \langle p \rangle (\psi * f)(h)$. By the semantics of sequents, f is universally quantified and can be instantiated arbitrarily.

3 POSIX Specification

This section defines the state, operations and invariants of the POSIX ASM.

3.1 Data Structures

The file system state consists of a directory tree t, a file store fs, and a registry of open file handles oh. Files are referenced by file identifiers of the abstract sort *Fid*. Open files are referenced by natural numbers ("file descriptors" in Unix).

$$\textbf{state vars} \quad t : \mathit{Tree}, \quad \mathit{fs} : \mathit{Fid} \rightharpoonup \mathit{FData}, \quad \mathit{oh} : \mathbb{N} \rightharpoonup \mathit{Handle}$$

The directory tree is specified as an algebraic data type *Tree* with two constructors: File nodes (`fnode`) form the leaves and store the identifier of the corresponding file. Directory nodes (`dnode`) make up the internal nodes and store the directory entries as a mapping from names to the respective subtrees.

$$\begin{aligned} \textbf{data}\ \mathit{Tree} \;=\; & \texttt{fnode}(\texttt{fid} : \mathit{Fid}) \\ \mid\; & \texttt{dnode}(\texttt{meta} : \mathit{Meta},\ \texttt{entries} : \mathit{String} \rightharpoonup \mathit{Tree}) \end{aligned}$$

The test $t.\texttt{isdir}$ yields whether tree t is a `dnode`. The abstract sort *Meta* is a placeholder for any further associated information. We postulate some selectors for $md : \mathit{Meta}$, to retrieve for example read, write and execute permissions $\texttt{pr}(u, md), \texttt{pw}(u, md), \texttt{px}(u, md)$ for some unspecified user $u : \mathit{User}$. This formalization of permissions has been taken from [12].

Files are given by the data type *FData* that stores the content as a list of bytes, and—analogously to directories—some associated metadata.

$$\textbf{data}\ \mathit{FData}\ =\ \texttt{fdata}(\texttt{meta} : \mathit{Meta}, \texttt{content} : \mathit{List}\langle \mathit{Byte}\rangle)$$

File handles store a file identifier `fid`, and keep track of the current read/write offset `pos` in bytes, and a mode, which can be read-only, write-only or read-write.

$$\begin{aligned}\textbf{data}\ \ &\mathit{Handle}\ =\ \texttt{handle}(\texttt{fid} : \mathit{Fid}, \texttt{pos} : \mathbb{N}, \texttt{mode} : \mathit{Mode})\\ \textbf{data}\ \ &\mathit{Mode}\ =\ \texttt{r} \mid \texttt{w} \mid \texttt{rw}\end{aligned}$$

The initial state is given by an empty root directory and no files:

$$\textbf{initial state}\quad t = \texttt{dnode}(md, \emptyset) \wedge \mathit{fs} = \emptyset \wedge \mathit{oh} = \emptyset \tag{3}$$

A path p is valid in a directory tree t, denoted by $p \in t$, if starting from the root t the path can be followed recursively such that each component is mapped by the respective subdirectory. Validity is defined by structural recursion over the path, where ϵ denotes the empty path, s/p denotes a path starting with component $s : \mathit{String}$ and remainder p, and $\epsilon \in t$ always holds.

$$\begin{aligned}s/p \in \texttt{fnode}(\mathit{fid}) &\ \leftrightarrow\ \texttt{false}\\ s/p \in \texttt{dnode}(md, st) &\ \leftrightarrow\ (s \in st \wedge p \in st[s])\end{aligned}$$

Lookup of a valid path p retrieves the respective subtree of t, denoted by $t[p]$. It is defined similarly to validity of paths:

$$t[\epsilon] = t \qquad \text{and} \qquad \texttt{dnode}(md, st)[s/p] = st[s][p] \quad \text{if } s \in st$$

It follows that validity of paths is prefix-closed, i.e., if $p/p' \in t$ then $p \in t$, furthermore $t[p]$ is a directory node if $p' \neq \epsilon$.

The expression $t[p/s \rightsquigarrow t']$ denotes the tree t with an additional subtree t' at path p/s. It is only specified for $p \in t$, i.e., it only adds the last component of the path to the tree. A converse function $t - p$ denotes the tree t without the whole subtree at path p. It is only specified for $p \in t$. Note that both modifications are non-destructive and construct new trees. Let the assignment $t[p] \coloneqq t'$ abbreviate $t \coloneqq t[p \rightsquigarrow t']$, analogously to function update.

Validity, lookup, insertion and deletion of paths compose with $\_/\_$, e.g.:

$$p/p' \in t \quad \leftrightarrow \quad (t \in p \wedge p' \in t[p]) \qquad \text{and} \qquad t[p/p'] = t[p][p'] \quad \text{if } p \in t$$

3.2 Operations & Error Handling

Operations realize the POSIX specification by using the algebraic functions on trees. Additionally, they perform extensive *error checks* to guard the file system against unintended or malicious calls to operations. Specifically, all operations are total (defined for all possible values of input parameters). Nevertheless, we use the term "precondition" to characterize valid inputs such that an operation succeeds. Violation of preconditions leads to an error *without* modifying the internal state. The POSIX model is presented in Fig. 4—omitting some generic error handling code at the beginning of each operation, which we explain by means of the create operation fully shown in Fig. 2.

Error handling is nondeterministic. It is possible that two errors conditions occur simultaneously, e.g., the whole path does not exist, or permissions to traverse an existing prefix are violated. The POSIX model does not restrict the order in which different conjuncts of preconditions are checked. Preconditions are defined as predicates $\texttt{pre-}op(in, err)$ that specify possible error codes for an input *in* given to the operation op. An implementation just has to satisfy the constraints imposed by these predicates.

Technically, an appropriate error code err' is nondeterministically chosen and assigned to the output variable *err*. If the operation is determined to succeed (implying a valid input) the body of `posix_create` picks a fresh file identifier *fid* for the new file, updates the directory tree with the corresponding file node and extends the file store by an empty file with the given initial metadata *md*. The operation is visualized in Fig. 3. The grey subtree corresponds to the parent directory $t[\texttt{parent}(p)]$; the newly created file node and associated data are denoted by the dashed triangle and box respectively.

Precondition-predicates contribute a significant part of the specification. They are defined by case distinction on possible error codes, as shown below. Certain errors, such as hardware failure or memory allocation (denoted by $\texttt{EIO}, \ldots$) are not restricted, i.e. they may occur anytime. Note that an implementation must thus recover from such situations to the previous abstract state.

$$\texttt{pre-create}(p, md, t, fs, e)$$

$$\leftrightarrow \quad \begin{cases} p \notin t \wedge \texttt{parent}(p) \in t \wedge t[\texttt{parent}(p)].\texttt{isdir}, & \text{if } e = \texttt{ESUCCESS} \\ p \in t, & \text{if } e = \texttt{EEXIST} \\ \texttt{true}, & \text{if } e \in \{\texttt{EIO}, \ldots\} \\ \ldots \end{cases}$$

The ASM code relies on several helpers that operate on lists: $\texttt{resize}(len; l)$ adjusts the size of list l to *len*, possibly padding l with zeroes at the end; `copy` and `splice` copy *len* elements of the source list *src* starting from offset *spos* into list *dst* at offset *dpos*. The latter operation corresponds exactly to the semantics of the POSIX write operation, i.e., it may extend *dst* at the end as shown below. The length of a list l is denoted by $\#\, l$.

$\texttt{splice}(src, spos, dpos, len; dst)$

```
if len ≠ 0 then
  if dpos + len < # dst then resize(dpos + len; dst)
  copy(src, spos, dpos, len; dst)
```

As a further twist the operations `posix_read` and `posix_write` may actually process less than *len* bytes and still succeed, either because the concrete implementation runs out of disk-space during the write, or due to an intermediate low-level error. This is modeled by `choose` n `with` $n < len$ `in` $len := n$.

Handles may refer to files that are not referenced by the tree any more, subsequently called *orphans*. This facility is actually exploited by applications to hide temporary files (e.g, MySQL caches, Apache locks) and during system/package updates.

3.3 Invariants

The POSIX state t, fs, oh has two explicit invariants. The easy one is simply that the root must be a directory (t.`isdir`). The second invariant states that the set of file identifiers referenced by t or oh is equal to `dom`(fs). It guarantees that for any *fid* in use, the associated file data in *fs* is available, and that *fs* contains no garbage. Given an overloaded function

$$\texttt{fids} : \mathit{Tree} \to \mathit{Multiset}\langle \mathit{Fid} \rangle \qquad \texttt{fids} : (\mathbb{N} \nrightarrow \mathit{Handle}) \to \mathit{Set}\langle \mathit{Fid} \rangle$$

the invariant can be defined formally:

$$\textbf{invariant} \quad \texttt{dom}(fs) = \underbrace{\{\mathit{fid} \mid \mathit{fid} \in \texttt{fids}(t)\} \cup \texttt{fids}(oh)}_{\texttt{fids}(t, oh)} \tag{4}$$

with $\texttt{fids}(oh) = \{oh[n].\texttt{fid} \mid n \in oh\}$ and

$$\texttt{fids}(\texttt{fnode}(\mathit{fid})) = \{\mathit{fid}\} \qquad \texttt{fids}(\texttt{dnode}(md, st)) = \biguplus_{s \in st} \texttt{fids}(st[s])$$

where $\uplus$ denotes multiset sum. Multisets are preferred over ordinary sets for the file identifiers in the tree for two reasons. On one hand, the number of occurrences of *fid* in the set `fids`(t) correlates to the number of hard links to a file. On the other hand, the effect of insertion or removal of a subtree on `fids` directly maps to $\uplus$ respectively $\setminus$. The proofs for invariant (4) are straightforward. The critical operations are `unlink`, `rename`, and `close`, that need to check whether the last link was removed and delete the file content if so.

3.4 Related Work

There exist several file system models with different scope and data structures, with a degree of abstraction similar to our POSIX model. These approaches typically make strong simplifications.

```
posix_create(p, md; err)
  choose err′
  with pre-create(p, md, t, fs, err′)
  in err := err′

  if err = ESUCCESS then
    choose fid with fid ∉ fs
    in t[p]   := fnode(fid)
       fs[fid] := fdata(md, ⟨⟩)
```

Fig. 2. POSIX create operation

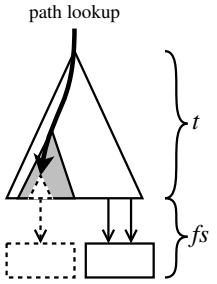

Fig. 3. FS tree

```
posix_mkdir(p, md; err)
  t[p] := dnode(md, ∅)

posix_rmdir(p; err)
  t := t − p

posix_link(p1, p2; err)
  let fid = t[p1].fid
  in  t[p2] := fnode(fid)

posix_unlink(p; err)
  let fid = t[p].fid
  in  t := t − p
      if fid ∉ fids(t, oh)
      then fs := fs − fid

posix_rename(p1, p2; err)
  let t1 = t[p1],  t2 = t[p2]
      ex = p2 ∈ t
  in  t := t − p2
      t[p2] := t1
      if  ex ∧ ¬ t2.isdir
         ∧ t2.fid ∉ fids(t, oh)
      then fs := fs − t2.fid

posix_truncate(p, len; err)
  let fid = t[p].fid
  in resize(len; fs[fid].content)

posix_readdir(p; names, err)
  names := dom(t[p].entries)

posix_open(p, mode; fd, err)
  let fid = t[p].fid
  in  choose n with n ∉ oh
  in  fd     := n
      oh[fd] := handle(fid, 0, mode)

posix_close(fd; err)
  let fid = oh[fd].fid
  in  oh := oh − fd
      if fid ∉ fids(t, oh)
      then fs := fs − fid

posix_read(fd; buf, len, err)
  let fid = oh[fd].fid
      pos = oh[fd].pos
  in  choose n with n ≤ len
      in len := n
      copy(fs[fid].content, pos, 0, len; buf)
      oh[fd].pos := pos + len

posix_write(fd, buf; len, err)
  let fid = oh[fd].fid
      pos = oh[fd].pos
  in  choose n with n ≤ len
      in len := n
      splice(buf, 0, pos, len; fs[fid].content)
      oh[fd].pos := pos + len
```

Fig. 4. POSIX operations

The approach to formalize a POSIX file system with an algebraic tree has been used previously only by Heisel [11] to evaluate specification languages and specification reuse.

Two other approaches occur in related models. In [19,12,8] the file system is specified as a mapping from paths to directories and files. This comes at the cost of an extra invariant that path validity is prefix-closed, which holds by construction in our model. However, only Hesselink and Lali [12] actually verify that it is preserved by operations.

Of these three models, only the one of Morgan and Sufrin [19] supports hard-links, using file identifiers as we do. In [12], equivalence classes of paths are suggested as an alternative solution. We are not aware of an attempt to realize this idea, though it would be interesting.

Damchoom et al. [5], in contrast, formalize the hierarchical structure by parent pointers with an acyclicity invariant. Hard links are inherently not supported by this design. We think that this approach is too different from the intuitive understanding of a file system to serve as top-level specification.

Morgan and Sufrin's work [19] contains a minor error (also found in [17,9]): they do not specify an equivalent of the test $len \neq 0$ in `splice` (Sec. 3.2), which may result in overly large files. The corresponding requirement in the POSIX standard [29] states that *[..] if nbytes [=len] is zero [..] the write() function shall return zero and have no other results.*

Open file handles have not been mechanized before, although these are specified on paper in [19], including the possibility of orphaned files.

Preconditions are treated similarly to [12], i.e., operations must not modify the state on errors. Ferreira et al [8] also have a comprehensive error specification in their POSIX-style specification, however, they fix the order of checks and allow arbitrary behavior on errors in their refinement proof obligations. To our knowledge, underspecified hardware failures are not admitted in related work.

4 VFS and AFS models

We give a short overview over the interplay between VFS and AFS and how generic file system aspects are separated from FS specific concerns. For details not covered here the reader is referred to [6] and the web presentation [7].

4.1 Interplay

The task of the VFS layer is to break down high-level POSIX operations to several calls of AFS operations. Fig. 5 visualizes a typical sequence for structural operations like `vfs_create`. In this case, it relies on three operations provided by the file system implementation, namely

1) lookup of the target of a single edge in the graph (`afs_lookup`),
2) retrieve the access permissions at each encountered node (`afs_iget`),
3) and finally the creation of the file.

Since many operations rely on *path lookup*, it is implemented as a subroutine `vfs_walk` that repeatedly performs steps 1) and 2). Figure 6 visualizes the representation of the file system state and effect of the operation. Analogously to Fig. 3, the parent directory is shaded in grey and the new parts are indicated by dashed lines. The cloud-shaped symbol summarizes the remaining directories and files.

The interface between VFS and AFS (resp. the concrete file system) is defined in terms of three communication data structures. *Inodes* ("Index Nodes") correspond to files and directories. They are identified by inode numbers $ino : Ino$ and store some metadata such as permissions but also size information and the number of hard-links. *Dentries* ("Directory Entries") correspond to the link between a parent directory and the inode of an entry. They contain the target inode number and a file/directory name. The content of files is partitioned into uniformly sized *pages*, which are sequences of bytes. A concrete implementation, as well as AFS, maps these to some internal state and on-disk structures. This approach decouples VFS from the file system, which is essential for modularity.

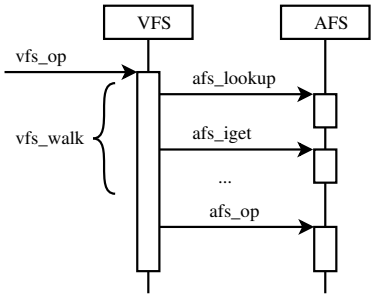

Fig. 5. VFS/AFS interplay

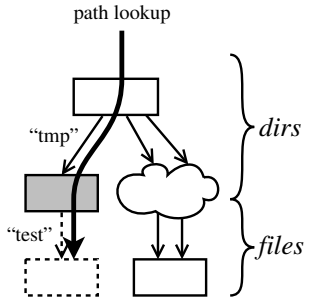

Fig. 6. FS as pointer structure

4.2 State

Although the VFS code is independent of the AFS state, its behavior is not. To define the abstraction relation (Sec. 5) and to prove the refinement (Sec. 6), we need to look into the AFS state, which is a pointer based acyclic graph with forward links. AFS keeps directories and files in two stores (partial functions) with disjoint domains, mapping inode numbers to the respective objects:

$$\textbf{state vars} \quad dirs : Ino \nrightarrow Dir, \quad files : Ino \nrightarrow File \qquad \text{where } Ino \simeq \mathbb{N}$$

The separation is motivated by the distinction into structural and content modifications: the former will affect mainly *dirs* while the latter will affect only *files*.

Directory entries are contained in the parent directory, likewise, pages are contained in the file object they belong to:

$$\begin{aligned}&\textbf{data } \mathit{Dir} \;\; = \texttt{dir}(\texttt{meta} : \mathit{Meta},\ \texttt{entries} : \mathit{String} \rightarrow\!\!\!\!\!\!\cdot\ \ \mathit{Ino})\\ &\textbf{data } \mathit{File} \;\; = \texttt{file}(\texttt{meta} : \mathit{Meta},\ \texttt{size} : \mathbb{N},\ \texttt{pages} : \mathbb{N} \rightarrow\!\!\!\!\!\!\cdot\ \ \mathit{Page}) \quad \text{where}\\ &\textbf{type } \mathit{Page} = \mathit{List}_{\texttt{PAGE\_SIZE}}\langle \mathit{Byte}\rangle\end{aligned}$$

Inode numbers $\mathit{ino} \in \mathit{dirs}$ or $\mathit{ino} \in \mathit{files}$ are called *allocated*, they refer to valid directories resp. files. The pages of a file need not to be contiguous, there may be holes in a file that implicitly contain zeroes. A function $\texttt{links}(\mathit{ino}, \mathit{dirs})$ returns a set of pairs $(\mathit{p\_ino}, \mathit{name})$ such that $\mathit{dirs}[\mathit{p\_ino}].\texttt{entries}[\mathit{name}] = \mathit{ino}$

The VFS state consists of open file handles that are equivalent to the ones in POSIX:

$$\begin{aligned}&\textbf{state var } \mathit{oh} : \mathbb{N} \rightarrow\!\!\!\!\!\!\cdot\ \ \mathit{Handle}, \qquad \text{where}\\ &\textbf{data } \mathit{Handle} = \ \texttt{handle}(\texttt{ino} : \mathit{Ino}, \texttt{pos} : \mathbb{N}, \texttt{mode} : \mathit{Mode})\end{aligned}$$

The initial state is given by an empty root directory with a fixed inode number `ROOT_INO` and no files:

$$\textbf{initial state} \quad \mathit{dirs} = [\texttt{ROOT\_INO} \mapsto \texttt{dir}(\mathit{md}, \emptyset)] \wedge \mathit{fs} = \emptyset \wedge \mathit{oh} = \emptyset \tag{5}$$

4.3 Operations

For each POSIX operation there is a corresponding VFS operation with the same signature, implementing the desired functionality, e.g. $\texttt{vfs\_create}(p, \mathit{md}; \mathit{err})$. Subroutine `vfs_walk` is shown in Fig. 7, as well as `afs_create`. Calls to `afs_iget` occur during the permission-check in `vfs_maylookup`. On success, `vfs_walk` establishes validity of a path q ($= \texttt{parent}(p)$ for `vfs_create`), expressed as

$$\text{vfs\_walk post:} \qquad \mathit{err} = \texttt{ESUCCESS} \rightarrow \texttt{path}(q, \mathit{p\_ino}, \mathit{ino}, \mathit{dirs}, \mathit{files})$$

The predicate `path` defined recursively on the path by the axioms

$$\begin{aligned}\texttt{path}(\epsilon, \mathit{p\_ino}, \mathit{ino}, \mathit{dirs}, \mathit{files}) \;&\leftrightarrow\; \mathit{p\_ino} = \mathit{ino} \wedge \mathit{ino} \in (\mathit{dirs} \cup \mathit{files})\\ \texttt{path}(s/p, \mathit{p\_ino}, \mathit{ino}, \mathit{dirs}, \mathit{files}) \;&\leftrightarrow\; \mathit{p\_ino} \in \mathit{dirs} \wedge s \in \mathit{si}\\ &\qquad \wedge \texttt{path}(p, \mathit{si}[s], \mathit{ino}, \mathit{dirs}, \mathit{files})\end{aligned}$$

where si abbreviates $\mathit{dirs}[\mathit{p\_ino}].\texttt{entries}$.
An overloaded version $\texttt{path}(p, \mathit{p\_ino}, \mathit{dirs}, \mathit{files})$ hides ino by existential quantification.

The VFS read and write operations break file access down to a number of individual page reads resp. writes in a loop (Fig. 8). It writes at most $\mathit{end} - \mathit{start}$ bytes of *buf* to the file specified by $\mathit{inode}.\texttt{ino}$, beginning at file offset *start* (in bytes); *total* counts the number of bytes already written.

The body of the write loop, `vfs_write_block` (Fig. 9), unifies partial writes to the first and last page alongside writes of entire pages, in order to avoid

```
vfs_walk(q; ino, err)
  err := ESUCCESS
  while q ≠ ϵ ∧ err = ESUCCESS do
    vfs_maylookup(ino; err)
    if err = ESUCCESS then
       let dent = negdentry(q.first)
       in  afs_lookup(ino; dent, err)
           if err = ESUCCESS then
              ino := dent.name
              q   := q.rest
```

```
afs_create(p_ino, md; dent, err)
  if   p_ino ∈ dirs
     ∧ dent.isnegdentry
     ∧ dent.name ∉ dirs[p_ino].entries
  then choose ino
  with ino ∉ dirs ∧ ino ∉ files ∧ ino ≠ 0
  in dirs[p_ino].entries[dent.name] := ino
     files[ino] := file(md, 0, ∅)
     dent       := dentry(dent.name, ino)
     err        := ESUCCESS
```

Fig. 7. VFS create operation and path walk

code for these special cases and intermediate assertions in the verification. The operation `afs_readpage` returns page number *pageno* of the file inode numbered *inode*.`ino` if that page is stored. Otherwise, an empty page is returned. The relevant part of *buf* is copied into the page and the page is written back.

Note that less than $end - start$ bytes may be written overall, since the loop is aborted as soon as an error is returned by AFS. Such an error is recovered by the test for $total \neq 0$ in Fig. 8, and—if necessary—the file size is adjusted to the number of bytes actually written.

4.4 Related Work

Galloway et al. [10] abstract the existing Linux VFS code to a SPIN model to check correct usage of locks and reference counters. Work with similar focus that directly checks the C source code is [20,31]. However, these approaches limit themselves to specific properties that are weaker than functional correctness (e.g., memory safety) or cover concepts orthogonal to this work (e.g., correct usage of locks).

Reading and writing files at byte-level has been addressed in [1,17]. We compare to this work in more detail in Sec. 6.3.

To our knowledge, our model [6] is the first one separating common functionality (VFS) and file system specific parts (AFS) with the goal of full functional verification.

5 Abstraction Relation

The abstraction relation `ABS` is defined as

$$\begin{aligned}&\texttt{ABS}(t, fs, oh_1, dirs, files, oh_2)\\ &\quad\leftrightarrow\quad fs = \texttt{fs}(files)\ \wedge\ \texttt{tree}(t, \texttt{ROOT\_INO})(dirs)\ \wedge\ oh_1 = oh_2\end{aligned}$$

where $\texttt{fs} : (Ino \nrightarrow File) \to (Fid \nrightarrow File)$ specifies the abstract file store *fs* and $\texttt{tree} : Tree \times Ino \to ((Ino \nrightarrow Dir) \to \mathbb{B})$ abstracts the pointer structure

```
...
let start = oh[fid].pos,  end = start + len,  total = 0,  done = false in
while ¬ done ∧ err = ESUCCESS do
  vfs_write_block(start, end, inode; done, buf, total, err)
if total ≠ 0 then err := ESUCCESS
if err = ESUCCESS ∧ inode.size < start + total then
  afs_truncate(inode.ino, start + total; err)
```

Fig. 8. VFS write operation (omitting error handling)

```
vfs_write_block(start, end, inode, buf, dirs; total, done, files, err)
  let pageno = (start + total) / PAGE_SIZE          // integer division
      offset = (start + total) % PAGE_SIZE          // and     modulo
      page   = emptypage
  in      // bytes to write in this iteration
      let n = min(end          − (start + total),   // write size     boundary
                  PAGE_SIZE − offset) in            // current page boundary
      if n ≠ 0 then
         afs_readpage(inode.ino, pageno, dirs, files; page, err)
         if err = ESUCCESS then
            copy(buf, total, offset, n; page)
            afs_writepage(inode.ino, pageno, page, dirs; files, err)
            total := total + n
      else done := true
```

Fig. 9. VFS code to write a partial page

with root `ROOT_INO` to the directory tree t using Separation Logic. By defining $\mathit{Fid} := \mathit{Ino}$, open file handles can be mapped by identity. This section formally defines `tree` and `fs` and states several key lemmas connecting the abstract and concrete states.

5.1 Directory Abstraction

The directory tree is mapped to the store of directories *dirs*, instantiating the separation logic theory from Sec. 2.2 with $\mathit{Loc} := \mathit{Ino}$ and $\mathit{Val} := \mathit{Dir}$. We define the predicate $\texttt{tree}(t, \mathit{ino})$ by structural recursion on the tree. The idea is that whenever $\texttt{tree}(t, \mathit{ino})(\mathit{dirs})$ holds, *ino* is the number of the root inode of a file system tree in *dirs* that corresponds to t.

$$\texttt{tree}(\texttt{fnode}(\mathit{fid}), \mathit{ino}) = \ (\texttt{emp} \wedge \mathit{ino} = \mathit{fid}) \tag{6}$$

$$\texttt{tree}(\texttt{dnode}(\mathit{md}, \mathit{st}), \mathit{ino}) = \tag{7}$$
$$\exists\ \mathit{si}.\ \texttt{dom}(\mathit{si}) = \texttt{dom}(\mathit{st}) \ \wedge \ \mathit{ino} \mapsto \texttt{dir}(\mathit{md}, \mathit{si}) \ * \ \circledast_{s \in \mathit{st}} \texttt{tree}(\mathit{st}[s], \mathit{si}[s])$$

Assertion (6) for file nodes requires that the inode number corresponds to the *fid* of the node and that the remaining part of the heap is empty.

Assertion (7) for directory nodes requires a corresponding directory in *dirs* that has the same metadata and corresponding directory entries *si*. The iterated separating conjunction $\circledast$ recursively asserts the abstraction relation for all subtrees $\mathit{st}[s]$ to children $\mathit{si}[s]$ in pairwise *disjoints* parts of *dirs*.

One can show by induction on p that $\texttt{tree}(t, \mathit{ino})(\mathit{dirs})$ implies

$$\texttt{path}(p, \mathit{ino}, \mathit{dirs}, \mathit{files}) \ \leftrightarrow \ p \in t$$

We furthermore define the assertion $\texttt{tree}|_p(t, \mathit{ino}_1, \mathit{ino}_2)(\mathit{dirs})$ that cuts out the subtree with root ino_2 at path p. Equality (8) encodes one main reasoning step for the proofs. It allows us to unfold the directory that is modified by an operation, given postcondition $\texttt{path}(p, \mathit{ino}_1, \mathit{ino}_2, \mathit{dirs}, \mathit{files})$ of `vfs_walk`

$$\texttt{tree}(t, \mathit{ino}_1)(\mathit{dirs}) \ \leftrightarrow \ (\texttt{tree}|_p(t, \mathit{ino}_1, \mathit{ino}_2) * \texttt{tree}(t[p], \mathit{ino}_2))(\mathit{dirs}) \tag{8}$$

Another critical lemma discards algebraic tree modifications if p is a (not necessarily strict) prefix of q:

$$q = p/p' \ \rightarrow \ \texttt{tree}|_p(t[q \rightsquigarrow t'], \mathit{ino}_1, \mathit{ino}_2) = \texttt{tree}|_p(t, \mathit{ino}_1, \mathit{ino}_2) \tag{9}$$

Finally, the abstraction implies the following equivalence, which ensures correct deletion of file content in `close`, `unlink` and `rename`:

$$\mathit{fid} \notin \texttt{fids}(t) \ \leftrightarrow \ \texttt{links}(\mathit{ino}, \mathit{dirs}) = \emptyset \quad \text{for } \mathit{ino} = \mathit{fid}$$

5.2 File Abstraction

The abstract file store is defined for each $\mathit{fid} \in \mathit{files}$, $\mathit{fid} = \mathit{ino}$ with $\mathit{files}[\mathit{ino}] = \texttt{file}(\mathit{md}, \mathit{size}, \mathit{pages})$ by the extensional equation

$$\texttt{fs}(\mathit{files})[\mathit{fid}] = \texttt{fdata}(\mathit{md},\ \texttt{content}(\mathit{pages})\ \texttt{to}\ \mathit{size})$$

where $\texttt{content} : (\mathbb{N} \rightarrow\!\!\!\!\!\!+\ \ \mathit{Page}) \to \mathit{Stream}\langle\mathit{Byte}\rangle$ assembles a stream of bytes from the pages of a file. The abstract file must store the finite prefix of length *size* of that stream. Streams $\sigma : \mathit{Stream}\langle\alpha\rangle$ can either be finite (a list) or infinite (a total function from natural numbers to values)

$$\textbf{type}\ \mathit{Stream}\langle\alpha\rangle = \mathit{List}\langle\alpha\rangle + (\mathbb{N} \to \alpha)$$

with a function $\#\,\sigma : \mathbb{N} + \{\infty\}$ to retrieve the length of a stream σ, prefix and postfix selectors $\sigma\ \texttt{to}\ n$ resp. $\sigma\ \texttt{from}\ n$ (defined for $n \leq \#\,\sigma$), and concatenation $\sigma_1 +\!\!+ \sigma_2$.

The abstraction to streams eliminates a lot of reasoning about list bounds and many case distinctions that would otherwise be necessary in definitions and proofs. In particular it simplifies the invariants of the loops in operations `vfs_read` and `vfs_write`, see Sec. 6.2.

We define the content of a file as an infinite stream with trailing zeroes beyond the end of the file:

$$\begin{aligned}\texttt{content}(\mathit{pages}) &= \lambda n.\ \texttt{getpage}(\mathit{pages}, n/\texttt{PAGE\_SIZE})[n\%\texttt{PAGE\_SIZE}]\\ \texttt{getpage}(\mathit{pages}, m) &= \text{if } m \in \mathit{pages} \text{ then } \mathit{pages}[m] \text{ else } \langle 0, \ldots\rangle\end{aligned}$$

6 Proofs

Proof obligation "initialization" (1) is trivial: (5) implies (3) for the same metadata *md* of the root directory and all invariants hold.

Proof obligation "correctness" (2) is established by symbolic execution of the VFS operation, which yields a state $\mathit{dirs}', \mathit{files}', \mathit{oh}'_1, \mathit{out}_1$, followed by symbolic execution of the POSIX operation to construct a matching witness run with a final state $t', \mathit{fs}', \mathit{oh}'_2, \mathit{out}_2$.

During symbolic execution, whenever the VFS chooses some value by the left rule for $\langle\!|\_|\!\rangle\_$ in (10), the POSIX is free to choose the *same* value by the existential quantifier in the right rule for $\langle\_\rangle\_$ in (10).

$$\frac{\vdash \forall x.\varphi(x) \to \langle\!|p|\!\rangle\psi \quad \vdash \exists x.\varphi(x)}{\vdash \langle\!|\texttt{choose}\ x\ \texttt{with}\ \varphi(x)\ \texttt{in}\ p|\!\rangle\psi} \qquad \frac{\vdash \exists x.\varphi(x) \wedge \langle p\rangle\psi}{\vdash \langle\texttt{choose}\ x\ \texttt{with}\ \varphi(x)\ \texttt{in}\ p\rangle\psi} \tag{10}$$

The error code err' selected by POSIX is determined this way, as well as e.g. the *fid* in in the operation `create` in Fig. 2 corresponding to the new inode number *ino* picked in Fig. 7.

The predicate logic goals resulting from symbolic execution have the form

$$\Gamma \vdash \mathtt{R}(t', \mathit{fs}', \mathit{oh}'_1, \mathit{dirs}', \mathit{files}', \mathit{oh}'_2) \wedge \mathit{out}_1 = \mathit{out}_2$$

where $\Gamma = \mathtt{R}(t, fs, oh_1, dirs, files, oh_2), \ldots$ contains the initial instance of the simulation relation, as well as preconditions and other information that has been gathered during symbolic execution (e.g., results of the tests in conditionals and subroutine postconditions). The goals reduce to two core proof obligations:

$$\begin{array}{ll} \text{directories:} & \mathtt{tree}(t, \mathtt{ROOT\_INO})(dirs), \Gamma \vdash \mathtt{tree}(t', \mathtt{ROOT\_INO})(dirs') \\ \text{files:} & fs = \mathtt{fs}(files), \Gamma \vdash fs' = \mathtt{fs}(files') \end{array}$$

6.1 Proof Strategy for Directories

Two types of modifications to the directory tree occur: insertions $t' = t[p \rightsquigarrow \ldots]$ and deletions $t' = t - p$ at a path p. These correspond to a local modification of some directory $dirs' = dirs[ino \mapsto \mathtt{dir}(md', si')]$ (for some new metadata md' and directory entries si') resp. $dirs' = dirs - ino$, where ino is found at $\mathtt{parent}(p)$.

The proof strategy is determined by the symbolic execution rules for assignment and deallocation. The notation $\psi_h^{h'}$ denotes renaming of the heap h to a fresh variable h' representing the updated heap in the remaining program modality resp. postcondition ψ.

$$\frac{(l \mapsto v * \varphi)(h') \vdash \psi_h^{h'}}{(l \mapsto_- * \varphi)(h) \vdash \langle h[l] := v \rangle \psi} \text{ assign-h} \qquad \frac{(\varphi)(h') \vdash \psi_h^{h'}}{(l \mapsto_- * \varphi)(h) \vdash \langle h := h - l \rangle \psi} \text{ dealloc}$$

The first step is to unfold the tree by (8) and (7) so that the maplet for ino is explicit and the assignment can be applied, propagating the assertion to the new directory store $dirs'$. The `dnode` predicate for ino is restored wrt. the new subdirectories si', e.g., by introducing an additional `fnode` assertion in the proof for `create`. The context $\mathtt{tree}|_p$ is rewritten to t' as well by (9) (applied from right to left), so that the whole abstraction can be folded by reverse-applying (8). Most of these steps are automated by rewrite rules.

6.2 Proof Strategy for Files

For $ino = fid$ and given $\sigma = \mathtt{content}(pages)$ the following two top-level equalities promote the concrete modification through the abstraction `fs`:

$$\mathtt{fs}(files[ino \mapsto \mathtt{file}(size, md, pages)]) = \mathtt{fs}(files)[fid \mapsto \mathtt{fdata}(md, \sigma \ \mathtt{to} \ size)]$$
$$\mathtt{fs}(files - ino) = \mathtt{fs}(files) - fid$$

It remains to establish that σ `to` $size$ matches the abstract operation, which is trivial for `create` ($\sigma \ \mathtt{to} \ 0 = \langle\rangle$) and difficult for `write` because of the loop in VFS, see Fig. 9.

The loop invariant for writing states that the file content can be decomposed into parts of the initial file $\mathtt{content}(pages_0)$ at the beginning and at the end,

with data from the buffer in between:

$$\begin{aligned} \text{write inv:} \quad \texttt{content}(\mathit{pages}) = \quad & \texttt{content}(\mathit{pages}_0)\ \texttt{to}\ \mathit{start} \\ & +\!\!+\ \mathit{buf}\ \texttt{to}\ \mathit{total} \\ & +\!\!+\ \texttt{content}(\mathit{pages}_0)\ \texttt{from}\ (\mathit{start} + \mathit{total}) \end{aligned} \tag{11}$$

The key idea behind the proofs to propagate the invariant through the loop is to normalize all terms of type stream to a representation with ++. For example, the effect of `afs_writepage` is captured by the equality

$$\begin{aligned} & \texttt{content}(\mathit{pages}[\mathit{pageno} \mapsto \mathit{page}]) \\ & \quad = \quad \texttt{content}(\mathit{pages})\ \texttt{to}\ (\mathit{pageno} * \texttt{PAGE\_SIZE}) \\ & \qquad +\!\!+\ \mathit{page} \\ & \qquad +\!\!+\ \texttt{content}(\mathit{pages})\ \texttt{from}\ (\mathit{pageno} * (\texttt{PAGE\_SIZE} + 1)) \end{aligned}$$

A similar theorem exists for $\texttt{copy}(\mathit{buf}, \mathit{total}, \mathit{offset}, n; \mathit{page})$. Equation (11) is then restored by distribution lemmas such as $(\sigma_1 +\!\!+ \sigma_2)\ \texttt{from}\ n = \sigma_2\ \texttt{from}\ (n - \#\,\sigma_1)$ if $n \geq \#\,\sigma_1$, and by cancellation of leading stream components of both sides of the equation $(\sigma +\!\!+ \sigma_1 = \sigma +\!\!+ \sigma_2) \leftrightarrow \sigma_1 = \sigma_2$ for finite σ. Finally, the loop invariant is mapped to the respective abstract POSIX operation.

Compared to the canonical alternative—a formulation of the loop invariants with `splice` (resp. `copy` for reading)—our approach is considerably more elegant: Invariant (11) does not need to mention the "current" size of the file, which would lead to case distinctions whether the file needs to grow. Such case distinctions (also found in `max`) produce a quadratic number of cases in the proof as one needs to consider the previous *and* the new version of the invariant.

6.3 Related Work

Hesselink and Lali [12] refine the mapping from paths to files to a pointer-based tree that is structurally similar to our AFS model. Their abstraction function is a point-wise comparison on path lookup. Our verification bridges a wider conceptual gap, since we start with a more abstract data structure (algebraic tree) and our VFS model is closer to a real implementation (e.g., uses a while-loop for path lookup, separates AFS). We have specified and verified additional operations, namely, access to files via file handles and the operations `read`, `write` and `truncate`.

Damchoom et al. [5] introduce several concepts such as the distinction between files and directories and permissions by small refinement steps. These aspects are covered by our POSIX model, except that their model is more detailed wrt. metadata (file owners, timestamps). Furthermore, in [4] the same authors decompose file write into parallel atomic updates of single pages, though not down to bytes.

Arkoudas et al. [1] address reading and writing of files in isolation (without file handles). Their model of file content is similar to ours (i.e., non-atomic

pages). They prove correctness of read and write with respect to an abstract POSIX-style specification. However, their file system interface allows only to access *single bytes* at a time, which is a considerable simplification.

The work of Kang and Jackson [17] is closest to our work with respect to read and write—it provides the same interface (buffer, offset, length). However, their model only deals with file content but not with directory trees or file handles. They check correctness with respect to an abstract specification for small bounded models. In comparison, their read and write algorithm is less practical than ours, because it relies on an explicit representation of a list of blocks that needs to be modified during an operation.

The VeriFast tool[2], which is based on Separation Logic, ships with some examples for binary trees, in particular, a solution to the VerifyThis competition[3] that specifies an equivalent to $\texttt{tree}|_p$ for binary trees. Finally, [18] is a nice application of Separation Logic to the verification of B^+ trees in the Coq prover.

7 Discussion and Conclusions

We have presented a formal specification of the POSIX file system interface, and a verified refinement to a formal model of a Virtual Filesystem Switch as a major step in the construction of a verified file system for Flash memory. As a consequence we can focus on the flash specific aspects in the future.

The different models have been developed more or less simultaneously in order to clarify the requirements for VFS, and to ensure that refinements will work out (the one presented in this paper as well as future ones).

We estimate that the net-effort put into this work was about six person-months: Understanding the POSIX requirements as well as the design of the Linux VFS and its source code took roughly one month. The remaining time was spent for design and specification of the models (about three months) and verification of invariants and refinement (two months). As a reference, we think that the verification was about three times as complex as the original Mondex challenge [27]. The size of the models is roughly as follows. The state machines consist of 50 lines in the POSIX model, 500 for VFS and 100 for AFS. Additionally, there are around 450 of lines algebraic specification for POSIX and 200 for VFS+AFS on top of the KIV libraries.

For this particular verification, state invariants were fairly easy to prove, while the simulation proofs were challenging. Paying attention to details (short read/write, orphans, errors) introduced additional complexity. We experienced that choosing the right data structures simplified both specification and verification (`fids` as multisets, file abstraction to streams).

Several orthogonal aspects remain for future work. Concurrency in VFS has been intentionally left out so far. Caching of inodes, dentries and pages in VFS could be realized without changing the AFS code. Fault tolerance against power loss is of great interest and we are currently proving that the models can deal

[2] `www.cs.kuleuven.be/~bartj/verifast/`

[3] `http://fm2012.verifythis.org`

with unexpected power loss anytime during the run of an operation, using the temporal program logic of KIV [26]. Translation of the models to C code is still an open issue.

References

1. K. Arkoudas, K. Zee, V. Kuncak, and M.C. Rinard. On verifying a file system implementation. In *Proc. of ICFEM*, pages 373–390, 2004.
2. E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15(1–2):237–257, 2003.
3. E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer, 2003.
4. K. Damchoom and M. Butler. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In M.V. Oliveira and J. Woodcock, editors, *Formal Methods: Foundations and Applications*, pages 134–152. Springer, 2009.
5. K. Damchoom, M. Butler, and J.-R. Abrial. Modelling and proof of a tree-structured file system in Event-B and Rodin. In *Proc. of ICFEM*, pages 25–44. Springer, 2008.
6. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. A Formal Model of a Virtual Filesystem Switch. In *Proc. of SSV*, pages 33–45, 2012.
7. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. KIV models and proofs of VFS and AFS, 2012. `http://www.informatik.uni-augsburg.de/swt/projects/flash.html`.
8. M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying Intel flash file system core specification. In *Modelling and Analysis in VDM: Proc. of the fourth VDM/Overture Workshop*, pages 54–71. School of Computing Science, Newcastle University, 2008. Technical Report CS-TR-1099.
9. L. Freitas, J. Woodcock, and Z. Fu. Posix file store in Z/Eves: An experiment in the verified software repository. *Sci. of Comp. Programming*, 74(4):238–257, 2009.
10. A. Galloway, G. Lüttgen, J.T. Mühlberg, and R.I. Siminiceanu. Model-checking the linux virtual file system. In *Proc. of VMCAI*, pages 74–88. Springer, 2009.
11. M. Heisel. Specification of the Unix File System: A Comparative Case Study. In *Proc. of AMAST*, Springer, pages 475–488, 1995.
12. W.H. Hesselink and M.I. Lali. Formalizing a hierarchical file system. *Formal Aspects of Computing*, 24(1):27–44, 2012.
13. C.A.R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
14. C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
15. A. Hunter. A brief introduction to the design of UBIFS. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`, 2008.
16. R. Joshi and G.J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2), June 2007.
17. E. Kang and D. Jackson. Formal Modelling and Analysis of a Flash Filesystem in Alloy. In *Proc. of ABZ*, pages 294–308. Springer, 2008.
18. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Proc. of POPL*, pages 237–248. ACM, 2010.

19. C. Morgan and B. Sufrin. Specification of the unix filing system. In *Specification case studies*, pages 91–140. Prentice Hall Ltd., Hertfordshire, UK, 1987.
20. J.T. Mühlberg and G. Lüttgen. Verifying compiled file system code. *Formal Aspecpts of Computing*, 24(3):375–391, 2012.
21. A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *Proc. of POPL*, pages 261–274. ACM, 2010.
22. G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference*, pages 4186–4199. IEEE Computer Society, 2005.
23. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II, pages 13 – 39. Kluwer, Dordrecht, 1998.
24. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS*, pages 55–74. IEEE Computer Society, 2002.
25. G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, 336(2–3):403–435, 2005.
26. G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. Interleaved programs and rely-guarantee reasoning with ITL. In *Proc. of TIME*, IEEE Computer Society, pages 99–106, 2011.
27. S. Stepney, D. Cooper, and J. Woodcock. AN ELECTRONIC PURSE Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory, 2000.
28. M. Szeredi. File system in user space. `http://fuse.sourceforge.net`.
29. The Open Group. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2008 Edition, 2008. `http://www.unix.org/version3/online.html` (login required).
30. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *Proc. of POPL*, pages 97–108. ACM, 2007.
31. J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proc. of OSDI*, pages 273–288. USENIX, 2004.