Verifying a file system implementation

Konstantine Arkoudas, Karen Zee, Viktor Kuncak, Martin Rinard

MIT Computer Science and AI Lab
{arkoudas,kkz,vkuncak,rinard}@csail.mit.edu

Abstract. We present a correctness proof for a basic file system implementation. This implementation contains key elements of standard Unix file systems such as inodes and fixed-size disk blocks. We prove the implementation correct by establishing a simulation relation between the specification of the file system (which models the file system as an abstract map from file names to sequences of bytes) and its implementation (which uses fixed-size disk blocks to store the contents of the files). We used the Athena proof system to represent and validate our proof. Our experience indicates that Athena's use of block-structured natural deduction, support for structural induction and proof abstraction, and seamless integration with high-performance automated theorem provers were essential to our ability to successfully manage a proof of this size.

1 Introduction

In this paper we explore the challenges of verifying the core operations of a Unix-like file system [28, 34]. We formalize the specification of the file system as a map from file names to sequences of bytes, then formalize an implementation that uses such standard file system data structures as inodes and fixed-sized disk blocks. We verify the correctness of the implementation by proving the existence of a simulation relation [16] between the specification and the implementation.

The proof is expressed and checked in Athena [1], a new interactive theorem-proving system based on denotational proof languages (DPLs [2]) for first-order logic with sorts and polymorphism. Athena uses a Fitch-style natural deduction calculus [32], formalized via the semantic abstraction of *assumption bases*. High-level reasoning idioms that are frequently encountered in common mathematical practice are directly available to the user. Athena also includes a higher-order functional language in the style of Scheme and ML and offers flexible mechanisms for expressing proof-search algorithms in a trusted manner (akin to the "tactics" and "tacticals" of LCF-like systems such as HOL [19] and Isabelle [30]). Block-structured natural deduction is used not only for writing proofs, but also for writing tactics. This is a novel feature of Athena; all other tactic languages we are aware of are based on sequent calculi. It proved to be remarkably useful: Tactics are much easier to write in this style, and are a great help in making proofs more modular.

The proof comprises 283 lemmas and theorems, and took 1.5 person-months of full-time work to complete. It consists of roughly 5,000 lines of Athena text, for an average of about 18 lines per lemma. It takes about 9 minutes to check on a high-end Pentium, for an average of 1.9 seconds per lemma. Athena seamlessly integrates cutting-edge automated theorem provers (ATPs) such as Vampire [35] and Spass [36] in order to mechanically prove tedious steps, leaving the user to focus on the interesting parts of the proof. Athena invokes Vampire and Spass over 2,000 times during the course of the proof. That the proof is still several thousand lines long reflects the sheer size of the problem. For instance, we needed to prove 12 invariants and there are 10 state-transforming operations, which translates to 120 lemmas for each invariant/operation pair (I, f), each guaranteeing that f preserves I. Most of these lemmas are non-trivial; many require induction, and several require a number of other auxiliary lemmas. Further complicating matters is the fact that we can show that some of these invariants are preserved only if we assume that certain other invariants hold. In these cases we must consider simultaneously the conjunction of several invariants. The resulting formulas are very long and have dozens of quantified variables. We believe that Athena's combination of natural deduction, versatile mechanisms for proof abstraction, and seamless incorporation of very efficient ATPs were crucial to our ability to successfully complete a proof effort of this scale.

To place our results in a broader context, consider that organizations rely on storage systems in general and file systems in particular to store critical persistent data. Because errors can cause the file system to lose this data, it is important for the implementation to be correct. The standard wisdom is that core system components such as file systems will always remain beyond the reach of full correctness proofs, leaving extensive testing—and the possibility of undetected residual errors—as the only option. Our results, however, suggest that correctness proofs for crucial system components (especially for the key algorithms and data structures at the heart of such components) may very well be within reach.

The remainder of the paper is structured as follows. In Section 2 we present an abstract specification of the file system. This specification hides the complexity of implementation-specific data structures such as inodes and data blocks by representing files simply as indexable sequences of bytes. Section 3 presents our model of the implementation of the file system. This implementation contains many more details, e.g., the mapping from file names to inodes, as well as the representation of file contents using sequences of non-contiguous data blocks that are dynamically allocated on the disk. Section 4 presents the statement of the correctness criterion. This criterion uses an abstraction function [16] that maps the state of the implementation to the state of the specification. Section 4 also sketches out the overall strategy of the proof. Section 5 addresses the key role that reachability invariants played in this project; these invariants express the fundamental data structure consistency properties needed for correct functioning of the file system implementation. Section 6 discusses the key role that Athena tactics played in making the proof more tractable. The convenience of using these proof tactics illustrates the advantages of a language that smoothly integrates natural deduction proofs and a Turing-complete higher-order proof-search language. Section 7 describes our experience in extending the basic file system with a form of access control. The last two sections discuss related work and present our main conclusions.

2 Abstract specification of the file system

The abstract specification of the file system uses two primitive sorts *Byte* and *FileID*, representing bytes and file identifiers, respectively. *File* is defined as a resizable array of *Byte*, introduced as a sort abbreviation: $File = RSArrayOf(Byte)$. The abstract state of the file system, *AbState*, is defined as a finite map from file identifiers (*FileID*) to file contents (*File*), i.e., $AbState = FMap(FileID, File)$. (See the corresponding technical report [7] for details on the Athena formalization of finite maps and resizable arrays. For a presentation of Athena's syntax and formal semantics see [9]; for further sample applications see [3,4,6,20].) We also introduce a distinguished element of *Byte*, called *fillByte*, which will be used to pad a file if a write is attempted at a position exceeding the file size.

We begin by giving the signature of the abstract read operation, ***read***:

$$\textbf{declare}\ \boldsymbol{read} : \mathit{FileID} \times \mathit{Nat} \times \mathit{AbState} \to \mathit{ReadResult}$$

Thus ***read*** takes a file identifier *fid*, an index i in the file, and an abstract file system state $\boldsymbol{s}$, and returns an element of *ReadResult*. The latter is defined as the following datatype:

$$\textbf{datatype}\ \mathit{ReadResult} = \mathit{EOF} \mid \mathit{Ok}(\mathit{Byte}) \mid \mathit{FileNotFound}$$

Therefore, the result of any ***read*** operation is either *EOF*, if the index is out of bounds; or *FileNotFound*, if the file does not exist; or, if all goes well, a value of the form $Ok(v)$ for some byte v, representing the content of file *fid* at position i. More precisely, the semantics of ***read*** are given by the following three axioms:

$$[AR_1]\ \forall\, \mathit{fid}\ i\ \boldsymbol{s}\,.\ \mathit{lookUp}\,(\mathit{fid}, \boldsymbol{s}) = \mathit{NONE} \Rightarrow \boldsymbol{read}(\mathit{fid}, i, \boldsymbol{s}) = \mathit{FileNotFound}$$

$$[AR_2]\ \forall\, \mathit{fid}\ i\ \boldsymbol{s}\ \mathit{file}\,.\ [\mathit{lookUp}\,(\mathit{fid}, \boldsymbol{s}) = \mathit{SOME}(\mathit{file}) \wedge \mathit{arrayLen}(\mathit{file}) \le i] \Rightarrow \boldsymbol{read}(\mathit{fid}, i, \boldsymbol{s}) = \mathit{EOF}$$

$$[AR_3]\ \forall\, \mathit{fid}\ i\ \boldsymbol{s}\ \mathit{file}\ v\,.\ [\mathit{lookUp}\,(\mathit{fid}, \boldsymbol{s}) = \mathit{SOME}(\mathit{file}) \wedge \mathit{arrayRead}(\mathit{file}, i) = \mathit{SOME}(v)] \Rightarrow \boldsymbol{read}(\mathit{fid}, i, \boldsymbol{s}) = \mathit{Ok}(v)$$

Using the equality conditions for finite maps and resizable arrays, we can prove the following extensionality theorem for abstract states:

$$\forall\, \boldsymbol{s}_1\ \boldsymbol{s}_2\,.\ \boldsymbol{s}_1 = \boldsymbol{s}_2 \Leftrightarrow [\forall\, \mathit{fid}\ i\,.\ \boldsymbol{read}(\mathit{fid}, i, \boldsymbol{s}_1) = \boldsymbol{read}(\mathit{fid}, i, \boldsymbol{s}_2)] \tag{1}$$

The abstract ***write*** has the following signature:

$$\textbf{declare}\ \boldsymbol{write} : \mathit{FileID} \times \mathit{Nat} \times \mathit{Byte} \times \mathit{AbState} \to \mathit{AbState}$$

This is the operation that defines state transitions in our file system. It takes as arguments a file identifier *fid*, an index i indicating a file position, a byte v

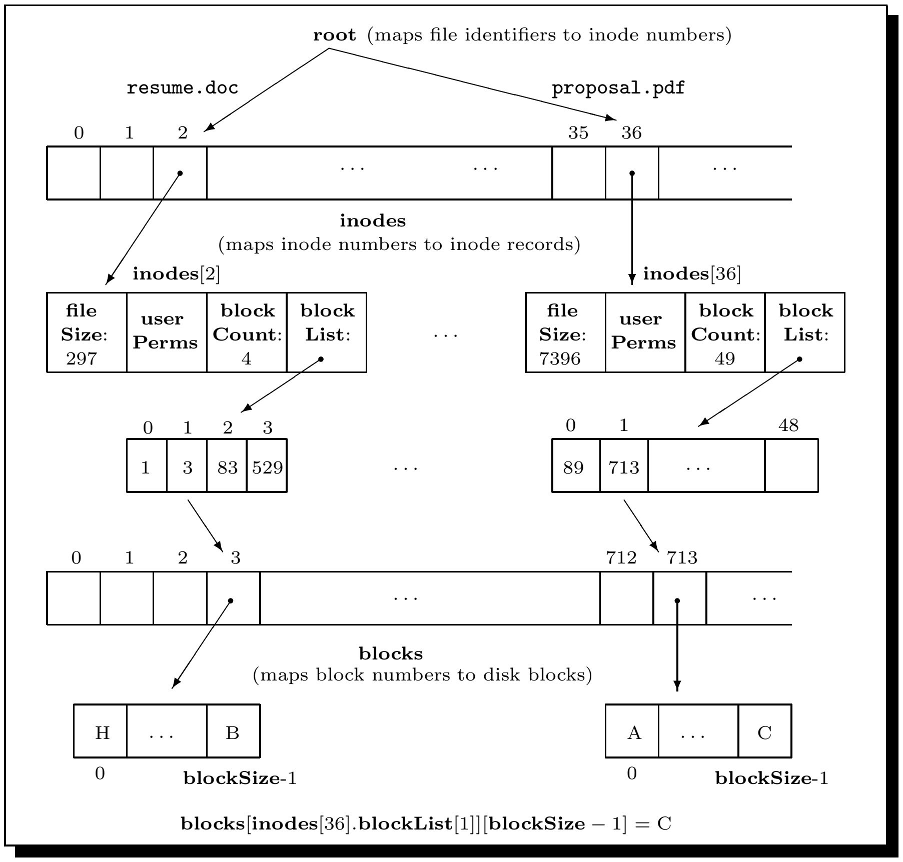

Fig. 1. A snapshot of the file system's state

representing the value to be written, and a file system state $\boldsymbol{s}$. The result is a new state where the file contents have been updated by storing v at position i. Note that if i exceeds the length of the file in state $\boldsymbol{s}$, then in the resulting state the file will be extended to size $i+1$ and all newly allocated positions below i will be padded with *fillByte*. Finally, if *fid* does not exist at all in $\boldsymbol{s}$, then an empty file of size $i+1$ is first created and then the value v is written. The axioms below make this precise:

$$[AW_1]\ \ \forall\, \mathit{fid}\ i\ v\ \boldsymbol{s}\,.\, \mathit{lookUp}\,(\mathit{fid}, \boldsymbol{s}) = \mathit{NONE} \Rightarrow$$
$$\boldsymbol{\mathit{write}}(\mathit{fid}, i, v, \boldsymbol{s}) = \mathit{update}(\boldsymbol{s}, \mathit{fid}, \mathit{arrayWrite}(\mathit{makeArray}(\mathit{fillByte}, i+1), i, v, \mathit{fillByte}))$$
$$[AW_2]\ \ \forall\, \mathit{fid}\ i\ v\ \boldsymbol{s}\ \mathit{file}\,.\, \mathit{lookUp}\,(\mathit{fid}, \boldsymbol{s}) = \mathit{SOME}(\mathit{file}) \Rightarrow$$
$$\boldsymbol{\mathit{write}}(\mathit{fid}, i, v, \boldsymbol{s}) = \mathit{update}(\boldsymbol{s}, \mathit{fid}, \mathit{arrayWrite}(\mathit{file}, i, v, \mathit{fillByte}))$$

3 File system implementation

Standard Unix file systems store the contents of each file in separate disk blocks, and maintain a table of structures called *inodes* that index those blocks and store various types of information about the file. Our implementation operates directly on the inodes and disk blocks and therefore models the operations that the file system performs on the disk. The file system organizes file data in *Block* units. Conceptually, a *Block* is an array of *blockSize* bytes, where *blockSize* is a positive constant. We model a *Block* as a finite map from natural numbers to bytes: $Block = FMap(Nat, Byte)$. A pictorial representation of a sample implementation state is shown in Figure 1. File meta-data is stored in inodes:

$$\textbf{datatype}\ INode = inode(fileSize : Nat, blockCount : Nat, blockList : FMap(Nat, Nat))$$

An *INode* is a datatype consisting of the file size in bytes and in blocks, and a list of block numbers. The list of block numbers is an array of the indices of the blocks that contain the file data. We model this array as a finite map from natural numbers (array indices) to natural numbers (block numbers).

The data type *State* represents the file system state:

$$\begin{gathered}\textbf{datatype}\ State = state(inodeCount : Nat, stateBlockCount : Nat,\\ inodes : FMap(Nat, INode), blocks : FMap(Nat, Block), root : FMap(FileID, Nat))\end{gathered}$$

A *State* consists of a count of the inodes in use; a count of the blocks in use; an array of inodes; an array of blocks; and the root directory. We model the array of inodes as a finite map from natural numbers (array indices) to *INode* (inodes). Likewise, we model the array of blocks as a finite map from natural numbers (array indices) to *Block* (blocks). We model the root directory as a finite map from *FileID* (file identifiers) to natural numbers (inode numbers).

```
proc extract ov = case ov of NONE -> fail | SOME v -> return v
proc read(fid,i) =
  case root[fid] of
  | NONE          -> return FileNotFound
  | SOME inodeNum ->
     var inode = extract(inodes[inodeNum]);
     if i >= inode.fileSize then return EOF
     else var blockNum = extract(inode.blockList[i div blockSize]);
          var block = blocks[blockNum];
          return extract(block[i mod blockSize])
```

Fig. 2. Pseudocode for the concrete read operation

The implementation of the fundamental file system operations in our model is presented in Figures 2 and 3 in pseudo code. In what follows we present the first-order axioms that correspond to these operations. We have manually derived these axioms from the pseudo code by explicitly threading the file system state through each operation.

3.1 Definition of the concrete read operation

The concrete read operation, *read* (Figure 2), has the following signature:

$$\textbf{declare}\ read : FileID \times Nat \times State \rightarrow ReadResult$$

```
proc write(fid,i,v) =
    case root[fid] of
    | NONE -> allocINode(fid);
              var inodeNum = root[fid];
              writeExisting(inodeNum,i,v)
   | SOME inodeNum -> writeExisting(inodeNum,i,v)

proc writeExisting(inodeNum,i,v) =
    var inode = extract(inodes[inodeNum]);
    if i div blockSize < inode.blockCount then
      if i < inode.fileSize then writeNoExtend(inodeNum,i,v)
      else writeSmallExtend(inodeNum,i,v)
    else extendFile(inodeNum,i);
         writeNoExtend(inodeNum,i,v)

proc writeSmallExtend(inodeNum,i,v) =
    var inode = extract(inodes[inodeNum]);
    var blockNum = extract(inode.blockList[i div blockSize]);
    var block = extract(blocks[blockNum]);
    inode.fileSize := i+1;
    inodes[inodeNum] := inode;
    block[i mod blockSize] := v;
    blocks[blockNum] := block

proc writeNoExtend(inodeNum,i,v) =
    var inode = extract(inodes[inodeNum]);
    var blockNum = extract(inode.blockList[i div blockSize]);
    var block = extract(blocks[blockNum]);
    block[i mod blockSize] := v;
    updateBlock(blockNum,block)

proc updateBlock(blockNum,block) = (blocks[blockNum] := block)

proc extendFile(inodeNum,i) =
    var inode = extract(inodes[inodeNum]);
    var blockIndex = i div blockSize;
    if blockIndex >= inode.blockCount then
      allocBlocks(inodeNum, blockIndex-inode.blockCount+1, i)

proc allocBlocks(inodeNum,toAlloc,i) =
    if toAlloc > 0 then
      getNextBlock();
      var inode = extract(inodes[inodeNum]);
      inode.fileSize := i+1;
      inode.blockList[inode.blockCount] := stateBlockCount-1;
      inode.blockCount := inode.blockCount+1;
      inodes[inodeNum] := inode;
      allocBlocks(inodeNum,toAlloc-1,i)

proc getNextBlock() = blocks[stateBlockCount] := initialBlock;
                      stateBlockCount := stateBlockCount+1

proc allocInode(fid) = root[fid] := inodeCount;
                       getNextInode()

proc getNextInode() = inodes[inodeCount] := initialInode;
                      inodeCount := inodeCount+1
```

Fig. 3. Pseudocode for the concrete write operation

(As a convention, we use bold italic font to indicate the abstract-state version of something: e.g., abstract ***read*** vs. concrete *read*, an abstract state ***s*** vs. a concrete state s, etc.) The *read* operation takes a file identifier *fid*, an index i in the file, and a concrete file system state s, and returns an element of *ReadResult*. It first determines if *fid* is present in the root directory of s. If not, *read* returns *FileNotFound*. Otherwise, it looks up the corresponding inode. If i is not less than the file size, *read* returns *EOF*. Otherwise, *read* looks up the block containing the data and returns the relevant byte. The following axioms capture these semantics (for ease of presentation, we omit universal quantifiers from now on; all variables can be assumed to be universally quantified):

$$[CR_1] \quad lookUp\,(fid, root(s)) = NONE \Rightarrow read(fid, i, s) = FileNotFound$$

$$\begin{gathered}[CR_2] \quad [lookUp\,(fid, root(s)) = SOME(n) \wedge \\ lookUp\,(n, inodes(s)) = SOME(inode(fs, bc, bl)) \wedge (fs \leq i)] \Rightarrow read(fid, i, s) = EOF\end{gathered}$$

$$\begin{gathered}[CR_3] \quad [lookUp\,(fid, root(s)) = SOME(n) \wedge \\ lookUp\,(n, inodes(s)) = SOME(inode(fs, bc, bl)) \wedge (i \ < \ fs) \wedge \\ lookUp\,(i \ div \ blockSize, bl) = SOME(bn) \wedge lookUp\,(bn, blocks(s)) = SOME(block) \wedge \\ lookUp\,(i \ mod \ blockSize, block) = SOME(v)] \Rightarrow read(fid, i, s) = Ok(v)\end{gathered}$$

3.2 Definition of the concrete write operation

The concrete write operation, *write* (Figure 3), takes a file identifier *fid*, a byte index i, the byte value v to write, and a state s, and returns the updated state:

$$\begin{gathered}\textbf{declare}\ write : FileID \times Nat \times Byte \times State \rightarrow State \\ [CW_1] \quad lookUp\,(fid, root(s)) = SOME(n) \Rightarrow write(fid, i, v, s) = writeExisting(n, i, v, s) \\ [CW_2] \quad \textbf{let}\ s' = allocINode(fid, s)\ \textbf{in} \\ [lookUp\,(fid, root(s)) = NONE \wedge lookUp\,(fid, root(s')) = SOME(n)] \Rightarrow \\ write(fid, i, v, s) = writeExisting(n, i, v, s')\end{gathered}$$

If the file associated with *fid* already exists, *write* delegates the write to the helper function *writeExisting*. If the file does not exist, *write* first invokes *allocINode*, which creates a new empty file by allocating and initializing an inode and then calls *writeExisting* with the inode number of the new file.

writeExisting takes an inode number n, a byte index i, the byte value v to write, and a state s, and returns the updated state:

$$\begin{gathered}\textbf{declare}\ writeExisting : Nat \times Nat \times Byte \times State \rightarrow State \\ [WE_1] \quad [lookUp\,(n, inodes(s)) = SOME(inode) \wedge \\ (i \ div \ blockSize) < blockCount(inode) \wedge i < fileSize(inode)] \Rightarrow \\ writeExisting(n, i, v, s) = writeNoExtend(n, i, v, s) \\ [WE_2] \quad [lookUp\,(n, inodes(s)) = SOME(inode) \wedge \\ (i \ div \ blockSize) < blockCount(inode) \wedge fileSize(inode) \leq i] \Rightarrow \\ writeExisting(n, i, v, s) = writeSmallExtend(n, i, v, s) \\ [WE_3] \quad [lookUp\,(n, inodes(s)) = SOME(inode) \wedge \\ blockCount(inode) \leq (i \ div \ blockSize)] \Rightarrow \\ writeExisting(n, i, v, s) = writeNoExtend(n, i, v, extendFile(n, i, s))\end{gathered}$$

If i is less than the file size, *writeExisting* delegates the writing to the helper function *writeNoExtend*, which stores the value v in the appropriate location.

If i is not less than the file size but is located in the last block of the file, *writeExisting* delegates to *writeSmallExtend*, which stores the value v in the appropriate position and updates the file size. Otherwise, *writeExisting* first invokes *extendFile*, which extends the file by the appropriate number of blocks, and then calls *writeNoExtend* on the updated state.[1]

extendFile takes an inode number n, the byte index of the write, and the state s. It delegates the task of allocating the necessary blocks to *allocBlocks*:

$$\begin{gathered}\textbf{declare}\ \mathit{extendFile} : \mathit{Nat} \times \mathit{Nat} \times \mathit{State} \rightarrow \mathit{State}\\ [\mathit{lookUp}\,(n, \mathit{inodes}(s)) = \mathit{SOME}(\mathit{inode}) \wedge \mathit{blockCount}(\mathit{inode}) \leq (j\ \mathit{div}\ \mathit{blockSize})] \Rightarrow\\ \mathit{extendFile}(n, j, s) = \mathit{allocBlocks}(n, (j\ \mathit{div}\ \mathit{blockSize}) - \mathit{blockCount}(\mathit{inode}) + 1, j, s)\end{gathered}$$

allocBlocks takes an inode number n, the number of blocks to allocate, the byte index j, and the state s. We define it by primitive recursion:

$$\begin{gathered}\textbf{declare}\ \mathit{allocBlocks} : \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \times \mathit{State} \rightarrow \mathit{State}\\ [\mathit{AB}_1] \quad \mathit{allocBlocks}(n, 0, j, s) = s\\ [\mathit{AB}_2] \quad [\mathit{getNextBlock}\,(s) = \mathit{state}(\mathit{inc}, \mathit{bc} + 1, \mathit{inm}, \mathit{bm}, \mathit{root}) \wedge\\ \mathit{lookUp}\,(n, \mathit{inm}) = \mathit{SOME}(\mathit{inode}(fs, \mathit{inbc}, \mathit{inbl}))] \Rightarrow\\ \mathit{allocBlocks}(n, k + 1, j, s) = \mathit{allocBlocks}(n, k, j, \mathit{state}(\mathit{inc}, \mathit{bc} + 1,\\ \mathit{update}(\mathit{inm}, n, \mathit{inode}(j + 1, \mathit{inbc} + 1, \mathit{update}(\mathit{inbl}, \mathit{inbc}, \mathit{bc})))))\end{gathered}$$

allocBlocks uses the helper function *getNextBlock*, which takes the state s, allocates and initializes the next free block, and returns the updated state:

$$\begin{gathered}\textbf{declare}\ \mathit{getNextBlock} : \mathit{State} \rightarrow \mathit{State}\\ \mathit{getNextBlock}\,(\mathit{state}(\mathit{inc}, \mathit{bc}, \mathit{inm}, \mathit{bm}, \mathit{root})) =\\ \mathit{state}(\mathit{inc}, \mathit{bc} + 1, \mathit{inm}, \mathit{update}(\mathit{bm}, \mathit{bc}, \mathit{initialBlock}), \mathit{root})\end{gathered}$$

Reachable states. In what follows we will restrict our attention to *reachable* states, those that can be obtained from the initial state by some finite sequence of *write* operations. Specifically, we define a predicate *reachableN* ("reachable in n steps") via two axioms: $\mathit{reachableN}(s, 0) \Leftrightarrow s = \mathit{initialState}$, and

$$\mathit{reachableN}(s, n + 1) \Leftrightarrow \exists\, s'\ \mathit{fid}\ i\ v\, .\, \mathit{reachableN}(s', n) \wedge s = \mathit{write}(\mathit{fid}, i, v, s')$$

We then set $\mathit{reachable}(s) \Leftrightarrow \exists\, n\, .\, \mathit{reachableN}(s, n)$. We will write $\widehat{\mathit{State}}$ for the set of all reachable states, and we will use the symbol $\widehat{s}$ to denote a reachable state. Propositions of the form $\forall \cdots \widehat{s} \cdots . P(\cdots \widehat{s} \cdots)$ and $\exists \cdots \widehat{s} \cdots . P(\cdots \widehat{s} \cdots)$ should be taken as abbreviations for $\forall \cdots s \cdots . \mathit{reachable}(s) \Rightarrow P(\cdots s \cdots)$ and $\exists \cdots s \cdots . \mathit{reachable}(s) \wedge P(\cdots s \cdots)$, respectively.

4 The correctness proof

4.1 State abstraction and homomorphic simulation

Consider the following binary relation A from concrete to abstract states:

$$\forall\, s\ \boldsymbol{s} . A(s, \boldsymbol{s}) \Leftrightarrow [\forall\, \mathit{fid}\ i\, .\, \mathit{read}(\mathit{fid}, i, s) = \boldsymbol{\mathit{read}}(\mathit{fid}, i, \boldsymbol{s})]$$

[1] We omit the axiomatization of *writeNoExtend* and a few other functions due to space constraints. The entire formalization along with the complete proof and a more detailed technical report [7] can be found at `www.cag.lcs.mit.edu/~kostas/dpls/athena/fs`.

$$\begin{tikzcd}
\mathit{FileID} \times \mathit{Nat} \times \widehat{\mathit{State}} \arrow[dd, "i \times i \times \alpha"] \arrow[dr, "\mathit{read}"] & \\
& \mathit{ReadResult} \\
\mathit{FileID} \times \mathit{Nat} \times \mathit{AbState} \arrow[ur, "\boldsymbol{read}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{FileID} \times \mathit{Nat} \times \mathit{Byte} \times \widehat{\mathit{State}} \arrow[r, "\mathit{write}"] \arrow[d, "i \times i \times i \times \alpha"] & \widehat{\mathit{State}} \arrow[d, "\alpha"] \\
\mathit{FileID} \times \mathit{Nat} \times \mathit{Byte} \times \mathit{AbState} \arrow[r, "\boldsymbol{write}"] & \mathit{AbState}
\end{tikzcd}$$

Fig. 4. Commuting diagrams for the *read* and *write* operations.

It follows directly from the extensionality principle on abstract states (1) that A is functional, i.e., that $\forall\, s\ \boldsymbol{s}_1\ \boldsymbol{s}_2 \,.\, A(s, \boldsymbol{s}_1) \wedge A(s, \boldsymbol{s}_2) \Rightarrow \boldsymbol{s}_1 = \boldsymbol{s}_2$. Accordingly, we postulate the existence of an abstraction function $\alpha : \mathit{State} \rightarrow \mathit{AbState}$ such that $\forall\, \widehat{s}\ \boldsymbol{s} \,.\, \alpha(\widehat{s}) = \boldsymbol{s} \Leftrightarrow A(\widehat{s}, \boldsymbol{s})$. That is, an abstracted state $\alpha(\widehat{s})$ has the exact same contents as $\widehat{s}$: reading any position of a file in one state yields the same result as reading that position of the file in the other state. Note that α, although total, is underspecified; it is only defined for reachable states.

A standard way of formalizing the requirement that an implementation $\mathcal{I}$ is faithful to a specification $\mathcal{S}$ is to express $\mathcal{I}$ and $\mathcal{S}$ as many-sorted algebras and establish a homomorphism from one to the other. In our case the two algebras are $\mathcal{I} = (\mathit{FileID}, \mathit{Nat}, \mathit{Byte}, \widehat{\mathit{State}}; \mathit{read}, \mathit{write})$ and

$$\mathcal{S} = (\mathit{FileID}, \mathit{Nat}, \mathit{Byte}, \mathit{AbState}; \boldsymbol{read}, \boldsymbol{write})$$

The embeddings from $\mathcal{I}$ to $\mathcal{S}$ for the carriers *FileID*, *Nat*, and *Byte* are simply the identity functions on these domains; while the embedding from $\widehat{\mathit{State}}$ to *AbState* is the abstraction mapping α. In order to prove that this translation yields a homomorphism we need to show that the two diagrams shown in Figure 4 commute. Symbolically, we need to prove the following:

$$\forall\, \mathit{fid}\ i\ \widehat{s} \,.\, \mathit{read}(\mathit{fid}, i, \widehat{s}) = \boldsymbol{read}(\mathit{fid}, i, \alpha(\widehat{s})) \tag{2}$$

and

$$\forall\, \mathit{fid}\ i\ v\ \widehat{s} \,.\, \alpha(\mathit{write}(\mathit{fid}, i, v, \widehat{s})) = \boldsymbol{write}(\mathit{fid}, i, v, \alpha(\widehat{s})) \tag{3}$$

4.2 Proof outline

Goal (2) follows immediately from the definition of the abstraction function α. For (3), since the consequent is equality between two abstract states and we have already proven that two abstract states $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ are equal iff any abstract read operation yields identical results on $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$, we transform (3) into the following:

$$\forall\, \mathit{fid}\ i\ v\ \widehat{s}\ \mathit{fid}'\ j \,.\, \boldsymbol{read}(\mathit{fid}', j, \alpha(\mathit{write}(\mathit{fid}, i, v, \widehat{s}))) = \boldsymbol{read}(\mathit{fid}', j, \boldsymbol{write}(\mathit{fid}, i, v, \alpha(\widehat{s})))$$

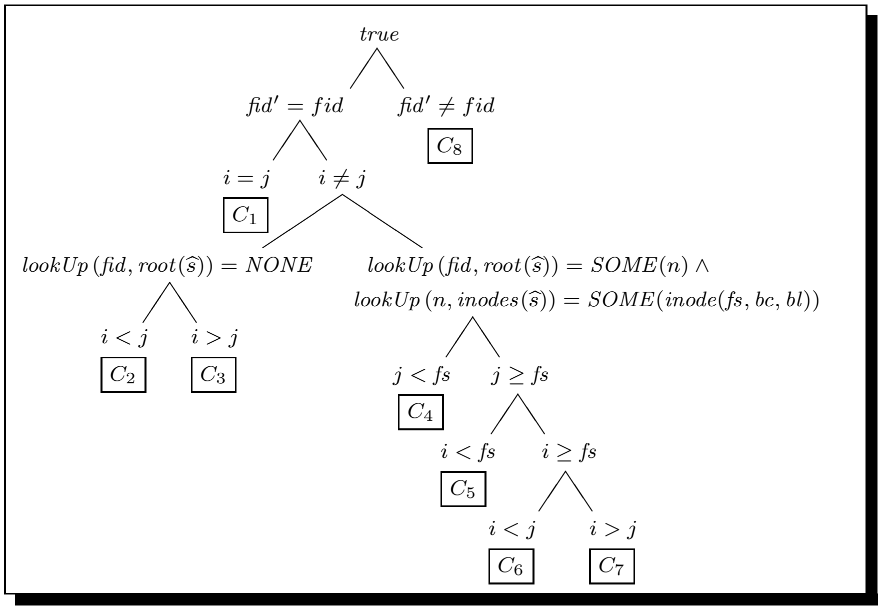

Fig. 5. Case analysis for proving the correctness of *write*.

Finally, using (2) on the above gives:

$$\forall\, fid\ fid'\ i\ j\ v\ \widehat{s}\,.\ read(fid', j, write(fid, i, v, \widehat{s})) = \boldsymbol{read}(fid', j, \boldsymbol{write}(fid, i, v, \alpha(\widehat{s})))$$

Therefore, choosing arbitrary fid, fid', j, v, i, and $\widehat{s}$, we need to show $L = R$, where $L = read(fid', j, write(fid, i, v, \widehat{s}))$ and

$$R = \boldsymbol{read}(fid', j, \boldsymbol{write}(fid, i, v, \alpha(\widehat{s})))$$

Showing $L = R$ is the main goal of the proof. We proceed by a case analysis as shown in Fig. 5. The decision tree of Fig. 5 has the following property: if the conditions that appear on a path from the root of the tree to an internal node u are all true, then the conditions at the children of u are mutually exclusive and jointly exhaustive (given that certain invariants hold, as discussed in Section 5). There are ultimately 8 distinct cases to be considered, C_1 through C_8, appearing at the leaves of the tree. Exactly one of those 8 cases must be true for any given $fid, fid', j, v, \widehat{s}$ and i. We prove that $L = R$ in all 8 cases.

For each case C_i, $i = 1, \ldots, 8$, we formulate and prove a pair of lemmas M_i and $\boldsymbol{M}_i$ that facilitate the proof of the goal $L = R$. Specifically, for each case C_i there are two possibilities: First, $L = R$ follows because both L and R reduce to a common term t, with $L = t$ following by virtue of lemma M_i and $R = t$ following by virtue of lemma $\boldsymbol{M}_i$: $L \xrightarrow{M_i} t \xleftarrow{\boldsymbol{M}_i} R$. Or, second, the identity follows because L and R respectively reduce to $read(fid', i, \widehat{s})$ and $\boldsymbol{read}(fid', i, \alpha(\widehat{s}))$, which are equal owing to (2). In that case, M_i is used to show $L = read(fid', i, \widehat{s})$ and $\boldsymbol{M}_i$ is used to show $R = \boldsymbol{read}(fid', i, \alpha(\widehat{s}))$:

$$L \xrightarrow{M_i} read(fid', i, \widehat{s}) = \boldsymbol{read}(fid', i, \alpha(\widehat{s})) \xleftarrow{\boldsymbol{M}_i} R \tag{4}$$

$$[M_1]\ \ read(fid, i, write(fid, i, v, \widehat{s})) = Ok(v)$$
$$[\boldsymbol{M}_1]\ \ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, i, v, \boldsymbol{s})) = Ok(v)$$

$$[M_2]\ \ [lookUp\,(fid, root(\widehat{s})) = NONE \wedge i < j] \Rightarrow read(fid, i, write(fid, j, v, \widehat{s})) = Ok(v)$$
$$[\boldsymbol{M}_2]\ \ [lookUp\,(fid, \boldsymbol{s}) = NONE \wedge i < j] \Rightarrow \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = Ok(v)$$

$$[M_3]\ \ [lookUp\,(fid, root(\widehat{s})) = NONE \wedge j < i] \Rightarrow read(fid, i, write(fid, j, v, \widehat{s})) = EOF$$
$$[\boldsymbol{M}_3]\ \ [lookUp\,(fid, \boldsymbol{s}) = NONE \wedge j < i] \Rightarrow \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = EOF$$

$$[M_4]\ \ \begin{array}{c}[lookUp\,(fid, root(\widehat{s})) = SOME(n) \wedge \\ lookUp\,(n, inodes(\widehat{s})) = SOME(inode(fs, bc, bl)) \wedge i \neq j \wedge j < fs] \Rightarrow \\ read(fid, i, write(fid, j, v, \widehat{s})) = read(fid, i, \widehat{s})\end{array}$$
$$[\boldsymbol{M}_4]\ \ \begin{array}{c}[lookUp\,(fid, \boldsymbol{s}) = SOME(A) \wedge i \neq j \wedge j < arrayLen(A)] \Rightarrow \\ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = \boldsymbol{read}(fid, i, \boldsymbol{s})\end{array}$$

$$[M_5]\ \ \begin{array}{c}[lookUp\,(fid, root(\widehat{s})) = SOME(n) \wedge \\ lookUp\,(n, inodes(\widehat{s})) = SOME(inode(fs, bc, bl)) \wedge fs \leq j \wedge i < fs] \Rightarrow \\ read(fid, i, write(fid, j, v, \widehat{s})) = read(fid, i, \widehat{s})\end{array}$$
$$[\boldsymbol{M}_5]\ \ \begin{array}{c}[lookUp\,(fid, \boldsymbol{s}) = SOME(A) \wedge arrayLen(A) \leq j \wedge i < arrayLen(A)] \Rightarrow \\ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = \boldsymbol{read}(fid, i, \boldsymbol{s})\end{array}$$

$$[M_6]\ \ \begin{array}{c}[lookUp\,(fid, root(\widehat{s})) = SOME(n) \wedge \\ lookUp\,(n, inodes(\widehat{s})) = SOME(inode(fs, bc, bl)) \wedge fs \leq i \wedge i < j] \Rightarrow \\ read(fid, i, write(fid, j, v, \widehat{s})) = Ok(fillByte)\end{array}$$
$$[\boldsymbol{M}_6]\ \ \begin{array}{c}[lookUp\,(fid, \boldsymbol{s}) = SOME(A) \wedge arrayLen(A) \leq j \wedge arrayLen(A) \leq i \wedge i < j] \Rightarrow \\ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = Ok(fillByte)\end{array}$$

$$[M_7]\ \ \begin{array}{c}[lookUp\,(fid, root(\widehat{s})) = SOME(n) \wedge \\ lookUp\,(n, inodes(\widehat{s})) = SOME(inode(fs, bc, bl)) \wedge fs \leq j \wedge j < i] \Rightarrow \\ read(fid, i, write(fid, j, v, \widehat{s})) = EOF\end{array}$$
$$[\boldsymbol{M}_7]\ \ \begin{array}{c}[lookUp\,(fid, \boldsymbol{s}) = SOME(A) \wedge arrayLen(A) \leq j \wedge arrayLen(A) \leq i \wedge j < i] \Rightarrow \\ \boldsymbol{read}(fid, i, \boldsymbol{write}(fid, j, v, \boldsymbol{s})) = EOF\end{array}$$

$$[M_8]\ \ fid_1 \neq fid_2 \Rightarrow read(fid_2, i, write(fid_1, j, v, \widehat{s})) = read(fid_2, i, \widehat{s})$$
$$[\boldsymbol{M}_8]\ \ fid_1 \neq fid_2 \Rightarrow \boldsymbol{read}(fid_2, i, \boldsymbol{write}(fid_1, j, v, \boldsymbol{s})) = \boldsymbol{read}(fid_2, i, \boldsymbol{s})$$

Fig. 6. Main lemmas

The eight pairs of lemmas are shown in Figure 6. The "abstract-state" versions of the lemmas ($[\boldsymbol{M}_i], i = 1, \ldots, 8$) are readily proved with the aid of Vampire from the axiomatizations of maps, resizable arrays, options, natural numbers, etc., and the specification axioms. The concrete lemmas M_i are much more challenging.

5 Proving reachability invariants

Reachable states have a number of properties that make them "well-behaved." For instance, if a file identifier is bound in the root of a state s to some inode number n, then we expect n to be bound in the mapping $inodes(s)$. While this is not true for arbitrary states s, it is true for reachable states. In what follows, by a state *invariant* we will mean a unary predicate on states $I(s)$ that is true for all reachable states, i.e., such that $\forall\, \widehat{s}\,.\, I(\widehat{s})$.

There are 12 invariants $inv_0, \ldots, inv_{11}$, that are of particular interest. The proof relies on them explicitly, i.e., at various points in the course of the argument we assume that all reachable states have these properties. Therefore, for the proof to be complete, we need to discharge these assumptions by *proving* that the properties in question are indeed invariants.

The process of guessing useful invariants—and then, more importantly, trying to prove them—was very helpful in strengthening our understanding of the implementation. More than once we conjectured false invariants, properties that appeared reasonable at first glance but later, when we tried to prove them, turned out to be false. For instance, a seemingly sensible "size invariant" is that for every inode of size *fs* and block count *bc* we have

$$fs = [(bc - 1) \cdot blockSize] + (fs \ mod \ blockSize)$$

But this equality does not hold when the file size is a multiple of the block count. The proper invariant is

$$\begin{gathered}[fs \ mod \ blockSize = 0 \Rightarrow fs = bc \cdot blockSize] \wedge \\ [fs \ mod \ blockSize \neq 0 \Rightarrow fs = ((bc - 1) \cdot blockSize) + (fs \ mod \ blockSize)]\end{gathered}$$

For any inode of file size *fs* and block count *bc*, we will write $szInv(fs, bc)$ to indicate that *fs* and *bc* are related as shown by the above formula. The following are the first four of the twelve invariants:

$$\begin{aligned}
&inv_0(s) : \forall \, fid \ n \, . \, [lookUp\,(fid, root(s)) = SOME(n)] \Rightarrow inDom(n, inodes(s)) \\
&inv_1(s) : \forall \, n \ fs \ bc \ bl \, . \, [lookUp\,(n, inodes(s)) = SOME(inode(fs, bc, bl))] \Rightarrow \\
&\qquad\qquad [inDom(n, inodes(s)) \Leftrightarrow n < bc] \\
&inv_2(s) \text{:} \ \forall \, n \ inode \ bn \ bn' \, . \, [lookUp\,(n, inodes(s)) = SOME(inode) \wedge \\
&\qquad lookUp\,(bn, blockList(inode)) = SOME(bn')] \Rightarrow inDom(bn', blocks(s)) \\
&inv_3(s) : \forall \, n \ fs \ bc \ bl \, . \, [lookUp\,(n, inodes(s)) = SOME(inode(fs, bc, bl))] \Rightarrow szInv(fs, bc)
\end{aligned}$$

These four invariants are fundamental and must be established before anything non-trivial can be proven about the system. They are also codependent, meaning that in order to prove that an operation preserves one of these properties we need to assume that the incoming state has all four of them. For the complete list of reachability invariants, see [7].

Showing that a unary state property $I(s)$ is an invariant proceeds in two steps. First, proving that I holds for the initial state; and second, proving $\forall \, fid \ i \ v \ s \, . \, I(s) \Rightarrow I(write(fid, i, v, s))$. Once both of these have been established, a routine induction on n will show $\forall \, n \ s \, . \, reachableN(s, n) \Rightarrow I(s)$. It then follows directly by the definition of reachability that all reachable states have I.

Proving that the initial state has an invariant inv_j is straightforward: in all 12 cases it is done automatically. The second step, proving that *write* preserves inv_j, is more involved. Including *write*, the implementation comprises 10 state-transforming operations,[2] and control may flow from *write* to any one of them.

[2] By a "state-transforming operation" we mean one that takes a state as an argument and produces a state as output. There are ten such operations, nine of which are auxiliary functions (such as *extendFile*) invoked by *write*.

Accordingly, we need to show that all ten operations preserve the invariant under consideration. This means that for a total of 10 operations $f_0, \ldots, f_9$ and 12 invariants $inv_0, \ldots, inv_{11}$, we need to prove 120 lemmas, each stating that f_i preserves inv_j.

The large majority of the proof text (about 80% of it) is devoted to proving these lemmas. Some of them are surprisingly tricky to prove, and even those that are not particularly conceptually demanding can be challenging to manipulate, if for no other reason simply because of their volume. Given the size of the function preconditions and the size of the invariants (especially in those cases where we need to consider the conjunction of several invariants at once), an invariance lemma can span multiple pages of text. Proof goals of that scale test the limits even of cutting-edge ATPs. For instance, in the case of a proposition P that was several pages long (which arose in the proof of one of the invariance lemmas), Spass took over 10 minutes to prove the trivial goal $P \Rightarrow P'$, where P' was simply an alphabetically renamed copy of P (Vampire was not able to prove it at all, at least within 20 minutes). Heavily skolemizing the formula and blindly following the resolution procedure prevented these systems from recognizing the goal as trivial. By contrast, using Athena's native inference rules, the goal was derived instantaneously via the two-line deduction **assume** P **in claim** P', because Athena treats alphabetically equivalent propositions as identical and has an efficient implementation of proposition look-ups. This speaks to the need to have a variety of reasoning mechanisms available in a uniform integrated framework.

6 Proof automation with Athena methods

After proving a few invariance lemmas for some of the operations it became apparent that a large portion of the reasoning was the same in every case and could thus be factored away for reuse. Athena makes it easy to abstract concrete proofs into natural-deduction proof algorithms called *methods*. For every state-transforming operation f_i we wrote a "preserver" method P_i that takes an arbitrary invariant I as input (expressed as a unary function that takes a state and constructs an appropriate proposition) and attempts to prove the corresponding invariance lemma. P_i encapsulates all the generic reasoning involved in proving invariants for f_i. If any non-generic reasoning (specific to I) is additionally required, it is packaged into a proof continuation K and passed into P_i as a higher-order method argument. P_i can then invoke K at appropriate points within its body as needed. Similar methods for other functions made the overall proof substantially shorter—and easier to develop and to debug—than it would have been otherwise.

Proof programmability was useful in streamlining several other recurring patterns of reasoning, apart from dealing with invariants. A typical example is this: given a reachable state $\widehat{s}$, an inode number n such that $lookUp(n, inodes(\widehat{s})) = SOME(inode(fs, bc, bl))$, and an index $i < fs$, we often need to prove the existence of bn and $block$ such that $lookUp(i \ div \ blockSize, bl) = SOME(bn)$ and

$lookUp\,(bn, blocks(\widehat{s})) = SOME(block)$. The reasoning required for this involves the invocation of various reachable-state invariants and standard laws of arithmetic. We packaged this reasoning in a method *find-bn-block* that takes all the relevant quantities as inputs, assumes that the appropriate hypotheses are in the assumption base, and performs the appropriate inferences.

7 Extending the basic model

To test the extensibility of our proof and to better model real systems, we augmented our models with user permissions after we completed the proof. We associated with each abstract file the set of authorized users, and modified the read and write operations to check that a user has permission before executing a read or a write. In the implementation, we similarly stored the user-permission information in a list associated with each inode. In the resulting models, not only the read operation, but also the write operation may return an error condition. The error condition for write occurs when the user does not have the permission to write a file. We modified the simulation relation condition to imply that a concrete write returns an error whenever the abstract write returns an error, and returns the corresponding concrete state otherwise. We then updated the proof to verify the new simulation relation condition.

Our experience from adapting the existing proof in the presence of changes to both the specification and to the implementation is encouraging. The overall structure of the proof did not change: the key components are still simulation relation conditions and invariants on reachable states. We were able to reuse a significant number of lemmas with no changes whatsoever. This includes not only useful abstractions such as resizable arrays, but also some non-interference properties of state-transforming operations. Most of the other lemmas required small adaptations to accommodate the additional components in the abstract and concrete state. We have found that the use of selector functions on algebraic data types improves the maintainability of proofs because it avoids explicitly listing all state components. In addition, accessors functions reduce the number of quantified variables in lemmas, making them more amenable to ATPs.

Another important factor contributing to the maintainability of our proofs is the abstraction of proof steps using Athena methods: instead of updating several similar proofs, it was often sufficient to update a single generic proof method. The ability to write recursive proof methods was also useful in increasing the granularity of proofs steps, reducing the typical size of proofs. In particular, we were able to noticeably increase the performance and robustness of calls to first-order theorem provers by wrapping them into a natural-deduction theorem-proving method written in Athena.

8 Related work

Techniques for verifying the correct use of file system interfaces expressed as finite state machines are presented in [15,17]. In this paper we have addressed the more difficult problem of showing that the file system implementation conforms to its specification. Consequently, our proof obligations are stronger and we have resorted to more general deductive verification. Static analysis techniques that handle more complex data structures include predicate abstraction and shape analysis [10,25,33]. These approaches are promising for automating proofs of program properties, but have not been used so far to show full functional correctness, as we do here.

Alloy [21] is a specification language based on a first-order relational calculus that can be used to specify complex structural constraints [23]. It has been used to describe the directory structure of a file system, but without modeling read and write operations. The use of the Alloy Analyzer to find counterexamples is complementary to our proofs [5]. Model checkers have been applied to systems whose main difficulty stems from concurrency [14, 38]. In contrast, the main challenge in our study is the complexity of the data structures that are needed for the correct functioning of the system. Abstract models of file systems have also been developed in Z [39, Chapter 15] and MooZ [29]; security properties of a Unix file system are studied in [37, Chapter 10]; security aspects of a CVS server were analyzed in [13]. These models capture properties that are largely orthogonal to the correct manipulation of the data stored in the files, which is the focus of our work.

It is interesting to consider whether the verification burden would be lighter with a system such as PVS [31] or ACL2 [22] that makes heavy use of automatic decision procedures for combinations of first-order theories such as arrays, lists, linear arithmetic, etc. We note that our use of high-performance off-the-shelf ATPs already provides a considerable degree of automation. In our experience, both Vampire and Spass have proven quite effective in non-inductive reasoning about lists, arrays, etc., simply on the basis of first-order axiomatizations of these domains. Our experience supports a recent benchmark study by Armando et al. [8], which showed that a state-of-the-art paramodulation-based prover with a fair search strategy compares favorably with CVC [11] in reasoning about arrays with extensionality. To further automate the verification process, we are considering the use of program analyses [18,26,27,33] in conjunction with decision procedures for decidable logics [12, 24] to formulate the key lemmas in the proof, generate some of the reachability invariants, and prove the frame conditions.

9 Conclusions

We presented a correctness proof for the key operations (reading and writing) of a file system based on Unix implementations. We are not aware of any other file system verification attempts dealing with such strong properties as the simulation relation condition, for all possible sequences of file system operations and without a priori bounds on the number of files or their sizes. Despite the apparent simplicity of this particular specification and implementation, our proofs shed light on the general reasoning techniques that would be required in establishing full functional correctness for any file system. Our experience with extending the basic model with new features indicates that the approach can be adapted to model additional relevant aspects of the file system implementation. These results suggest that a combination of state-of-the art formal methods techniques greatly facilitates the deductive verification of crucial software infrastructure components such as file systems.

We have found Athena to be a powerful framework for carrying out a complex verification effort. Polymorphic sorts and datatypes allow for natural data modeling; strong support for structural induction facilitates inductive reasoning over such datatypes; a block-structured natural deduction format helps to make proofs more readable and writable; and the use of first-order logic allows for smooth integration with state-of-the-art first-order ATPs, keeping the proof steps at a high level of detail. Perhaps most importantly, the novel assumption-base semantics of Athena make it possible to formulate not only proofs but also tactics in true natural-deduction ("Fitch") style. A significant technical insight emerging from this project is that tactics in such a style can be an exceptionally practical and useful tool for making proofs shorter and more modular.

References

1. K. Arkoudas. Athena. `www.cag.csail.mit.edu/~kostas/dpls/athena`.
2. K. Arkoudas. Denotational Proof Languages. PhD dissertation, MIT, 2000.
3. K. Arkoudas. Specification, abduction, and proof. In *Second International Symposium on Automated Technology for Verification and Analysis*, Taiwan, October 2004.
4. K. Arkoudas and S. Bringsjord. Metareasoning for multi-agent epistemic logics. In *CLIMA V*, Lisbon, Portugal, September 2004.
5. K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating model checking and theorem proving for relational reasoning. In *7th International Seminar on Relational Methods in Computer Science*, 2003.
6. K. Arkoudas and M. Rinard. Deductive runtime certification. In *Proceedings of the 2004 Workshop on Runtime Verification*, Barcelona, Spain, April 2004.
7. K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. On verifying a file system implementation. Technical Report 946, MIT CSAIL, May 2004.
8. A. Armando, M. P. Bonacina, S. Ranise, M. Rusinowitch, and A. K. Sehgal. High-performance deduction for verification: a case study in the theory of arrays. In S. Autexier and H. Mantel, editors, *Notes of the Workshop on Verification, Third Federated Logic Conference (FLoC02)*, pages 103–112, 2002.
9. T. Arvizo. A virtual machine for a type-ω denotational proof language. Masters thesis, MIT, June 2002.
10. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.
11. C. W. Barrett, D. L. Dill, and A. Stump. A framework for cooperating decision procedures. In D. A. McAllester, editor, *17th CADE*, volume 1831, pages 79–98. Springer, 2000.
12. E. Börger, E. Gräedel, and Y. Gurevich. *The Classical Decision Problem.* Springer-Verlag, 1997.
13. A. D. Brucker, F. Rittinger, and B. Wolff. A cvs-server security architecture: Concepts and formal analysis. Technical Report 182, Institut für Informatik Albert-Ludwigs-Universität Freiburg, December 2003. HOL-Z distribution, `http://wailoa.informatik.uni-freiburg.de/holz/index.html`.
14. W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE TSE*, pages 498–520, July 1998.

15. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM PLDI*, 2002.
16. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
17. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.
18. P. Fradet and D. L. Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.
19. M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, Cambridge, England, 1993.
20. M. Hao. Using a denotational proof language to verify dataflow analyses. Masters thesis, MIT, September 2002.
21. D. Jackson. Alloy: a lightweight object modelling notation. *ACM TOSEM*, 11(2):256–290, 2002.
22. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.
23. S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *15th IEEE ASE*, 2000.
24. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.
25. V. Kuncak and M. Rinard. Boolean algebra of shape analysis constraints. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, 2004.
26. P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39:46–55, March 2004.
27. J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. ACM PLDI*, Atlanta, GA, June 1988.
28. M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.
29. S. R. L. Meira, A. L. C. Cavalcanti, and C. S. Santos. The unix filing system: A MooZ specification. In K. Lano and H. Haughton, editors, *Object-oriented Specification Case Studies*, chapter 4, pages 80–109. Prentice-Hall, 1994.
30. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
31. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. SRI International, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park CA 94025.
32. F. J. Pelletier. A Brief History of Natural Deduction. *History and Philosophy of Logic*, 20:1–31, 1999.
33. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
34. K. Thompson. UNIX implementation. *The Bell System Technical Journal*, 57(6 (part 2)), 1978.
35. A. Voronkov et al. The anatomy of Vampire (implementing bottom-up procedures with code trees). JAR, 15(2):237–265, 1995.
36. C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.
37. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universitaet Muenchen, 2002.
38. J. M. Wing and M. Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, 28:273–299, 1997.
39. J. Woodcock and J. Davies. *Using Z*. Prentice-Hall, Inc., 1996.