Formalizing a Hierarchical File System

Wim H. Hesselink[1] M.I. Lali[2]

Dept. of Computing Science, University of Groningen
P.O.Box 407, 9700 AK Groningen, The Netherlands

Abstract

In this note, we define an abstract file system as a partial function from (absolute) paths to data. Such a file system determines the set of valid paths. It allows the file system to be read and written at a valid path, and it allows the system to be modified by the Unix operations for removal (*rm*), making of directories (*mkdir*), and moving (*mv*). We present abstract definitions (axioms) for these operations.
This specification is refined towards a pointer implementation. To mitigate the problems attached to partial functions, we do this in two steps. First a refinement towards a pointer implementation with total functions, followed by one that allows partial functions. These two refinements are proved correct by means of a number of invariants. Indeed, the insight gained mainly consists of the invariants of the pointer implementation that are needed for the refinement functions.
Finally, each of the three specification levels is enriched with a permission system for reading, writing, or executing, and the refinement relations between these permission systems are explored.

Keywords: File System, Specification, Verification, Refinement, Permission System, Theorem Proving.

1 Introduction

What is a hierarchical file system? Although most of us seem to know the answer, it is difficult to find a definition, let alone a specification. In [1], e.g., we read: “Like most modern operating systems, UNIX organizes its file system as a hierarchy of directories” and “directories, which contain information about a set of files and are used to locate a file by its name.” If this answers the question for the impatient, it does not yield a specification. Yet, a specification is needed when we want to verify the correctness of an implementation.

As file systems are at the core of the operating system kernel, even a simple error can cause a crash of the system, possibly resulting in loss of stored data [2]. File system errors are among the most dangerous errors because they can cause loss of persistent data stored on the disk. The growing size and complexity of file systems indicates the need of verification of such systems for ensuring reliability. It is very difficult to ensure reliability by testing techniques.

[1] Email:w.h.hesselink@rug.nl
[2] Email:m.i.ullah@rug.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Testing and simulation are traditional techniques to check that the software written is correct with respect to its functionality [3]. Many testing techniques are available which help in eliminating coding errors. However, very few defects in end products are due to coding errors. For example, in 197 critical faults, detected during the testing phase of the Voyager and Galileo spacecraft, just three of them were coding errors. About 50% of the faults were traced to requirements, 25% to design, and the rest due to other errors. This is a typical example of a prevalent problem that the majority of faults in software arise in requirements and design and very few occur due to coding. Furthermore, such techniques do not cover all possible behaviors of the system [4].

Formal verification uses the mathematical techniques for ensuring the design to conform to the functional correctness. It can be applied to designs described for many different levels of abstraction [5]. It helps in eliminating errors in the design which can cause disaster at later stages.

In this paper, we formalize the most rudimentary aspects of a hierarchical file system: only reading and writing files, deleting them, creating them, and moving them. We do this in a top-down fashion, starting with the point of view of a user who does not want to know anything of the implementation. This is refined into a version with directories that hold subdirectories.

When formalizing this, one encounters the problem of partial functions. In the first refinement step this is ignored by forcing the functions to be total. In the second refinement step, we recognize the inherent partiality of our functions. From the conceptual point of view, this may seem superfluous. For implementations, however, it is crucial because this partiality corresponds to the potential occurrence of unallocated pointers in the implementation.

We use the proof assistant PVS [6] for our formalization and the verification of the refinement relations. The PVS proof script of our definitions, theorems, and proofs is available at [7]. Our notation is partially based on PVS syntax, but we also use concepts from Haskell, and standard mathematical notations.

The primary contribution is to formally define a file system at a very high level with its five operations of reading and writing files, and creating, deleting and moving files and directories, and to refine this specification in two steps to a system with file identifiers as pointers, and to mechanically verify the refinement relations.

1.1 Related work

The 15 year old grand challenge in software verification proposed by Hoare in [8] was refined by Joshi and Holzmann in [9] to a mini-challenge to build a small verifiable file system for flash memory. The current status of the grand challenge is discussed in [10]. Earlier, in [11], C. Morgan and B. Sufrin proposed abstract specifications of some of the data structures in the UNIX file system. The POSIX file store using Z/Eves with refinements based on [11] is described in [12,13]. The paper [12] provides a concrete implementation of an abstract specification by means of Java HashMaps, taken from JML annotations given in [14]. Wenzel [15] analyses aspects of the Unix file system security with the proof assistant Isabelle/HOL. Galloway et al. [16] verify the existing Linux Virtual File System (VFS) using model checking techniques by extracting and validating a model from an available implementation of VFS. Yang et al. [2] build their own model checker "FiSC" to find serious file system errors. This paper shows that even the most popular file systems contain serious bugs which can cause damage to the stored data. Therefore, it is important to consider correctness proofs even of existing file system implementations. In this regard, a correctness proof of operations like reading and writing in a Unix based file system is presented in [17] using Athena, an interactive theorem-proving environment.

In 2008, inspired by Hughes' specification [18] of a visual file system in Z, Damchoom, Butler, and Abrial [19] have modeled a tree structured file system in Event-B and Rodin. This paper gives one of the first specifications of a hierarchical file system in which the tree structure can be modified. It is close to our work. An important difference, however, is that it is more abstract in the sense that it ignores file names and paths, which are central concepts in our specifications.

1.2 Overview

In section 2, we construct an abstract specification of a hierarchical file system based on the "user point of view". Section 3 contains the first refinement step towards a file system with pointers that are modelled as total functions. Section 4 presents the second refinement step to a system with pointers modelled as partial functions. In section 5, we indicate how file permissions as used in Unix can be specified in our set-up. Conclusions are drawn in section 6.

2 The User's Point of View

From the user's point of view, a file store associates a file or a directory to an absolute path. For simplicity, we do not distinguish files and directories, i.e., we allow a file to be associated to a directory. In some later refinement, we may want to make the distinction, e.g., by restricting the data associated to a directory.

A path is thus a finite sequence of (directory) names, and the type of paths is defined by

$$\mathit{Path} = \mathtt{finite\_sequence}[\mathit{Name}]\ .$$

A store determines the valid paths, and the associated data for each valid path. We therefore define an abstract store as a partial function from *Path* to *Data*, according to the following type definition:

$$\mathit{StoreA} = [\mathit{Path} \to \mathtt{lift}[\mathit{Data}]]\ ,$$

where we use the PVS definition $\mathtt{lift}[X] = X \cup \{\bot\}$. The set of valid paths for an abstract store x is given by

$$\mathit{Valid}(x) = \{p \mid x(p) \neq \bot\}\ .$$

We use the operator $+\!\!+$ for concatenation of paths as finite sequences. This operator is associative, i.e., $(p +\!\!+ q) +\!\!+ r = p +\!\!+ (q +\!\!+ r)$, and has the empty path ε as two-sided unit, i.e., $\varepsilon +\!\!+ p = p = p +\!\!+ \varepsilon$. Path p is called a *prefix* of q, with notation $p \sqsubseteq q$, iff there is a path r with $p +\!\!+ r = q$. Relation $\sqsubseteq$ is an ordering of the set *Path*, i.e., it is reflexive and transitive, and $p \sqsubseteq q \sqsubseteq p$ implies $p = q$.

The empty path ε holds the root of the file system and should therefore always be valid. A prefix of a valid path should be valid. We therefore define a store x to be *legitimate* if

$$\varepsilon \in \mathit{Valid}(x) \ \wedge \ (\forall\, p, q : p \sqsubseteq q \ \wedge \ q \in \mathit{Valid}(x) \ \Rightarrow \ p \in \mathit{Valid}(x)) \ .$$

Reading the data of a path p in store x is just asking for $x(p)$, which yields $\perp$ iff $p \notin \mathit{Valid}(x)$.

Writing a file means modifying the data according to some recipe, e.g., writing from a certain offset. Such a recipe can be regarded as an element of the type

$$\mathit{Modifier} = [\mathit{Data} \rightarrow \mathit{Data}] \ .$$

Writing with modifier m at path p in store x is only successful when p is valid; otherwise nothing happens. For simplicity, we do not yet include error messages for failure. We therefore lift every modifier m to $\mathtt{lift}[\mathit{Data}]$ by defining $m(\perp) = \perp$ and define writing by:

$$\begin{array}{l} \mathit{write} : [\mathit{Path} \times \mathit{Modifier} \times \mathit{StoreA} \rightarrow \mathit{StoreA}] \ , \\ \mathit{write}(p, m, x) = (x \ \mathbf{with} \ [(p) := m(x(p))]) \ , \\ \text{or equivalently: } \ \mathit{write}(p, m, x)(q) = (q = p \ ? \ m(x(p)) : x(q)) \ . \end{array}$$

Here we use the **with** notation of PVS for function modification, with a C-like conditional expression as an alternative. If x is legitimate, then $\mathit{write}(p, m, x)$ is also legitimate.

Remark. In an earlier version, the second argument of *write* was the new value for $x(p)$, of type *Data*. This was not expressive enough, because in actual file systems, writing often means replacing a part of the file or appending something to a file. All this can be expressed by means of modifiers. □

The Unix function *ls* associates to a given store x and a valid path p the set of names n that occur in the directory of p. We need to distinguish an empty directory from a nonexistent one. We therefore define:

$$\begin{array}{l} \mathit{ls} : [\mathit{Path} \times \mathit{StoreA} \rightarrow \mathtt{lift}[\mathbb{P}[\mathit{Name}]]] \ , \\ \mathit{ls}(p, x) = (p \in \mathit{Valid}(x) \ ? \ \{n \mid p \ \text{++}\ n \in \mathit{Valid}(x)\} : \ \perp) \ , \end{array}$$

where a name n is implicitly coerced to a singleton list. If the path is not valid, *ls* yields $\perp$.

We specify a function *create* that makes a new entry with data d in the store for a given path p. It does so only when path p is not yet valid and has a valid parent directory. Otherwise, *create* has no effect. Here, for a nonempty path p, the parent path $\mathit{parent}(p)$ is defined as the unique maximal strict prefix of p, which satisfies $|\mathit{parent}(p)| = |p| - 1$, where $|p|$ stands for the length of p.

$$\begin{array}{l} \mathit{create} : [\mathit{Path} \times \mathit{Data} \times \mathit{StoreA} \rightarrow \mathit{StoreA}] \ , \\ \mathit{create}(p, d, x) = \\ \quad (x(p) \neq \perp \ \vee \ x(\mathit{parent}(p)) = \perp \ ? \ x \ : \ x \ \mathbf{with} \ [(p) := d]) \ . \end{array}$$

If store x is legitimate, the store $y = \mathit{create}(p, d, x)$ is legitimate because $\mathit{Valid}(y) = \mathit{Valid}(x) \cup \{p\}$.

Deletion of a path p from an abstract store x also deletes all descendant directories. It is therefore specified by

$$\begin{aligned}&\mathit{deleteG} : [\mathit{Path} \times \mathit{StoreA} \to \mathit{StoreA}] \ , \\ &\mathit{deleteG}(p, x)(q) = (p \sqsubseteq q \ ? \perp : x(q)) \ .\end{aligned}$$

If store x is legitimate and $p \neq \varepsilon$, the store $y = \mathit{deleteG}(p, x)$ is legitimate because $\mathit{Valid}(y) = \mathit{Valid}(x) \setminus \{q \mid p \sqsubseteq q\}$.

Moving is more complicated. A move from p to q has the effect that the old directory q (if it was valid) is completely overwritten by p, whereas the old directory p disappears. Let store $y = \mathit{moveG}(p, q, x)$ be the result of the move. For a path r of the form $r = q \mathbin{+\!\!+} s$, we therefore have $y(r) = x(p \mathbin{+\!\!+} s)$. For $q \sqsubseteq r$, this implies $y(r) = x(p \mathbin{+\!\!+} \mathit{drop}(|q|, r))$ where $\mathit{drop}(k, r)$ is the suffix of r obtained by removing the first k elements. We thus obtain:

$$\begin{aligned}&\mathit{moveG} : [\mathit{Path} \times \mathit{Path} \times \mathit{StoreA} \to \mathit{StoreA}] \ , \\ &\mathit{moveG}(p, q, x)(r) = \\ &\quad (\ q \sqsubseteq r \ ? \ x(p \mathbin{+\!\!+} \mathit{drop}(|q|, r)) \\ &\quad : p \sqsubseteq r \ ? \perp \\ &\quad : x(r)\ ) \ .\end{aligned}$$

It is easy to see that $\mathit{moveG}(p, p, x) = x$ for any x and p. If store x is legitimate and $p \notin \mathit{Valid}(x)$, then $\mathit{moveG}(p, q, x) = \mathit{deleteG}(q, x)$.

Theorem 2.1 *Let x be a legitimate abstract store. Assume that $q \neq \varepsilon$ and $p \not\sqsubseteq \mathit{parent}(q)$, and that $\mathit{parent}(q) \in \mathit{Valid}(x)$. Then $\mathit{move}(p, q, x)$ is legitimate.*

Because of the case distinctions in the definition of *move*, the proof of this result is rather complicated. A key step in the proof is the observation that, if $q \sqsubseteq s$ and $r \sqsubseteq s$ and $q \not\sqsubseteq r$, then $r \sqsubseteq \mathit{parent}(q)$.

On the other hand, when $p \in \mathit{Valid}(x)$ is a strict prefix of q, then $y = \mathit{move}(p, q, x)$ satisfies $y(p) = \perp$ and $y(q) = x(p) \neq \perp$, so that store y is not legitimate.

We extended the names *deleteG* and *moveG* with G, because we need versions of these functions that preserve legitimacy. We thus define

$$\begin{aligned}&\mathit{delete}(p, x) = (p = \varepsilon \ ? \ x : \ \mathit{deleteG}(p, x)) \ , \\ &\mathit{move}(p, q, x) = (\ x(p) = \perp \vee q = \varepsilon \vee x(\mathit{parent}(q)) = \perp \vee p \sqsubseteq q \ ? \ x \\ &\qquad\qquad\qquad : \ \mathit{moveG}(p, q, x)\ ) \ .\end{aligned}$$

These functions *delete* and *move* indeed preserve legitimacy. With respect to *move*, we are slightly more restrictive than needed for Theorem 2.1. We let *move* do nothing if p is not valid or if p is a prefix of q itself, because moving is not useful if p is not valid or equal to q.

We finally specify an initial store with arbitrary data d and an empty directory:

$$\begin{aligned}&\mathit{initstoreA} : [\mathit{Data} \to \mathit{StoreA}] \ , \\ &\mathit{initstoreA}(d)(p) = (p = \varepsilon \ ? \ d : \perp) \ .\end{aligned}$$

It is easy to see that $\mathit{initstoreA}(d)$ is legitimate.

3 Refining the Store

The usual implementation of a file store is by means of the standard pointer implementation of a tree. We use a simple type *Fid* of file identifiers as the pointer type. The root of the tree is given by a constant $rootId \in Fid$. For now, we define a *directory* to be a total function that associates file identifiers to names. We use a constant $null \in Fid$ as a default file identifier for nonoccurring names. We postulate that $rootId \neq null$.

We thus allow nodes also for invalid paths. They always hold a directory, which may be empty, and they may have data. A *total store* is a total function from file identifiers to nodes.

$$\begin{aligned}&DirT = [Name \to Fid]\ ,\\ &NodeT = [\#\quad data : \mathtt{lift}[Data]\ ,\ dir : DirT\quad \#]\ ,\\ &StoreT = [Fid \to NodeT]\ .\end{aligned}$$

Here [# and #] are constructors for record types as used in PVS. The corresponding element constructors are (# and #) used below. For a node v, we write $v.data$ and $v.dir$ for its data and its directory. At this point, the nodes are more general than usual. Later on, we may want to impose conditions on the data for a node that contains a nonempty directory. A new node with data d and without children is declared by

$$nodeT(d) = (\#\ data := d\ ,\ dir := (\lambda\, n : null)\ \#)\ .$$

The initial store is defined by

$$initstoreT(d) = (\lambda\, f :\ f = rootId\ ?\ nodeT(d) : nodeT(\bot))\ .$$

Since a store x is supposed to be a total function, we postulate an invariant to ensure that no data are hidden in or beyond *null*, viz.

$J0(x):\quad x(null) = nodeT(\bot)\ .$

The file identifier associated to a path in a given store is defined recursively. For this purpose, we define a function $last : [Path \to Name]$ such that, for every nonempty path p, we have

$$p = parent(p)\ +\!\!+\ last(p)\ .$$

The file identifier of a path is given by the recursive *lookup* function L defined by:

$$\begin{aligned}&L : [Path \times StoreT \to Fid]\ ,\\ &L(p, x) = (\ p = \varepsilon\ ?\ \ rootId :\ \ x(L(parent(p), x)).dir(last(p))\ )\ .\end{aligned}$$

We only want to find $data = \bot$ at the node of *null*. This is expressed in the invariant

$J1(x):\quad \forall\, p : x(L(p, x)).data = \bot\ \Rightarrow\ L(p, x) = null\ .$

The abstraction function from total stores to abstract stores is defined by

$$\begin{aligned}&abstract : [StoreT \to StoreA]\ ,\\ &abstract(x)(p) = x(L(p, x)).data\ .\end{aligned}$$

It is straightforward to prove that $abstract(initstoreT(d)) = initstoreA(d)$. Using $J0(x)$ and $J1(x)$, one can easily prove

$$p \in \mathit{Valid}(\mathit{abstract}(x)) \quad \equiv \quad L(p,x) \neq \mathit{null} \ .$$

Using invariant $J0$, we prove that

$$L(p,x) = \mathit{null} \ \wedge \ p \sqsubseteq q \ \Rightarrow \ L(q,x) = \mathit{null} \ .$$

Using the postulate $\mathit{rootId} \neq \mathit{null}$, this implies that $\mathit{abstract}(x)$ is legitimate.

Reading is defined by

$$\mathit{read}(p,x) = \mathit{abstract}(x)(p) = x(L(p,x)).\mathit{data} \ .$$

The contents of a directory are found by means of function ls defined by

$$\mathit{ls}(p,x) = (L(p,x) = \mathit{null}\ ?\ \bot : \mathit{ls}(x(L(p,x)).\mathit{dir})) \ , \text{ where}$$
$$\mathit{ls}(di) = \{n \in \mathit{Name} \mid di(n) \neq \mathit{null}\} \ .$$

Using the invariants $J0$ and $J1$, it is easy to prove the refinement theorem that $\mathit{ls}(p, \mathit{abstract}(x)) = \mathit{ls}(p,x)$.

For writing, we use the PVS conventions for modifying functional structures. We thus define:

$$\begin{aligned} &\mathit{write}(p,m,x) = \\ &\quad (\ L(p,x) = \mathit{null}\ ?\ x \\ &\quad :\ x \ \mathbf{with}\ [(L(p,x)).\mathit{data} := m(x(L(p,x)).\mathit{data})]\ )\ . \end{aligned}$$

Writing does not change L, because writing affects only field data, while L only uses field dir. In other words, we have the easy result that

$$L(q, \mathit{write}(p,m,x)) = L(q,x) \ .$$

The specification of section 2 implies that writing at a path p only affects path p. This implies that the total store must be a tree, in the sense that different valid paths have different file identifiers. This is postulated in the invariant:

$$J2(x): \quad \forall\, p,q:\ L(p,x) = L(q,x) \neq \mathit{null} \ \Rightarrow \ p = q \ .$$

We now prove

Theorem 3.1 *Assume $J0(x)$, $J1(x)$, and $J2(x)$. Then we have* $\mathit{abstract}(\mathit{write}(p,m,x)) = \mathit{write}(p,m,\mathit{abstract}(x))$.

The challenge is now to define implementation functions for *create*, *delete*, and *move* that behave in the same way as the corresponding functions on *StoreA*, and to prove such facts.

3.1 Removals from the store

Given $x : \mathit{StoreT}$, a path p can only be deleted from it if it is not the root and it is valid. Deletion then amounts to removing its last name from its parent directory:

$$\begin{aligned} &\mathit{delete}(p,x) = \\ &\quad (\ p = \varepsilon\ ?\ x\ :\ x \ \mathbf{with}\ [\ (pp).\mathit{dir}(\mathit{last}(p)) := \mathit{null}\ ]\ ) \\ &\text{where} \ \ pp = L(\mathit{parent}(p), x). \end{aligned}$$

We postpone garbage collection to section 3.4.

It turns out that the invariants obtained above are enough to prove:

Theorem 3.2 *Assume that $J0(x)$ and $J2(x)$. Then we have $abstract(delete(p, x)) = delete(p, abstract(x))$.*

Proof. We first claim that

$$(0) \qquad L(q, delete(p, x)) = \\ (p \neq \varepsilon \ \wedge \ p \sqsubseteq q \ ?\ null : L(q, x)) \ .$$

This is proved by induction on the length of q, because L is defined recursively. The invariant $J2$ is needed because store x is modified at $pp.dir(last(p))$, and at several points we therefore need to ensure that the arguments we are interested in differ from this.

We verify the final step by observing for every path q:

$$\begin{array}{cl} & abstract(delete(p, x))(q) \\ = & \{ \text{ definition of } abstract\text{; write } y = delete(p, x) \ \} \\ & y(L(q, y)).data \\ = & \{ \ (0) \text{ and } J0 \text{ for } y \ \} \\ & (p \neq \varepsilon \ \wedge \ p \sqsubseteq q \ ?\ \bot \ : \ y(L(q, x)).data) \\ = & \{ \ x \text{ and } y \text{ are equal on } data \ \} \\ & (p \neq \varepsilon \ \wedge \ p \sqsubseteq q \ ?\ \bot \ : \ x(L(q, x)).data)) \\ = & \{ \text{ definitions of } delete \text{ and } abstract \ \} \\ & delete(p, abstract(x))(q) \ . \end{array}$$

This completes the proof. □

3.2 Creating new entries

In order to preserve *J2* when creating new entries in the store, we need an unbounded heap. We formally ensure this by postulating that the type *Fid* is infinite and that the stores we consider are all finite, according to the invariant

$$\begin{array}{ll} J3(x): & \#range(x) < \infty \text{ , where} \\ & range(x) = \{null, rootId\} \cup \{f \in Fid \mid \exists\, g, n : f = x(g).dir(n)\} \ . \end{array}$$

This enables us to define a choice function $new : StoreT \to Fid$ with the property:

$$(1) \qquad J3(x) \quad \Rightarrow \quad new(x) \notin range(x) \ .$$

Function *create* at this level of abstraction is defined by

$$\begin{array}{l} create(p, d, x) = \\ \quad (\ pp = null \ \vee \ L(x, p) \neq null \ ?\ x \\ \quad :\ x \ \mathbf{with} \ [\, (pp).dir(last(p)) := ln\, , \ (ln) := nodeT(d) \ ] \ ) \\ \text{where} \ \ pp = L(parent(p), x) \text{ and } ln = new(x). \end{array}$$

Function *create* satisfies the refinement theorem:

Theorem 3.3 *Assume that $J0(x) \wedge J2(x) \wedge J3(x)$. Then we have $abstract(create(p, d, x)) = create(p, d, abstract(x))$.*

Proof. One first proves that the failure conditions of both versions of *create* are equivalent, because $abstract(x)(q) = \bot$ if and only if $L(x, q) = null$. Now assume both versions modify the store. We then prove, by induction on the length of q, that

$$(2) \qquad \begin{array}{l} L(q, \mathit{create}(p,d,x)) = \\ \quad (\ q = p \neq \varepsilon \ \wedge \ L(\mathit{parent}(p),x) \neq \mathit{null} = L(p,x)\ ?\ \mathit{new}(x) \\ \quad :\ L(q,x)\ )\ . \end{array}$$

We verify the final step by observing for every path q:

$$\begin{array}{cl} & \mathit{abstract}(\mathit{create}(p,d,x))(q) \\ = & \{\ \text{definition of } \mathit{abstract};\ \text{write } y = \mathit{create}(p,d,x)\ \} \\ & y(L(q,y)).\mathit{data} \\ = & \{\ (2)\ \} \\ & (\ q = p \neq \varepsilon \ \wedge \ L(\mathit{parent}(p),x) \neq \mathit{null} = L(p,x)\ ?\ y(\mathit{new}(x)).\mathit{data} \\ & :\ y(L(q,x)).\mathit{data}) \\ = & \{\ \text{definition } y \text{ and } \mathit{new};\ L(q,x) \neq \mathit{new}(x)\ \} \\ & (\ q = p \neq \varepsilon \ \wedge \ L(\mathit{parent}(p),x) \neq \mathit{null} = L(p,x)\ ?\ d \\ & :\ x(L(q,x)).\mathit{data})) \\ = & \{\ \text{write } x' = \mathit{abstract}(x);\ \text{definition of } \mathit{abstract}\ \} \\ & (\ q = p \neq \varepsilon \ \wedge \ x'(\mathit{parent}(p)) \neq \bot = x'(p)\ ?\ d\ :\ x'(q)) \\ = & \{\ \text{abstract definition of } \mathit{create}\ \} \\ & \mathit{create}(p,d,x')(q)\ . \end{array}$$

This completes the proof. □

3.3 Moving files and directories

Function *move* at this level is defined by:

$$\begin{array}{l} \mathit{move}(p,q,x) = \\ \quad (\ q = \varepsilon \ \vee \ p \sqsubseteq q \ \vee \ L(p,x) = \mathit{null} \ \vee \ qq = \mathit{null}\ ?\ x \\ \quad :\ x \ \textbf{with}\ [(qq).\mathit{dir}(\mathit{last}(q)) := L(p,x)\ , \\ \qquad\qquad (pp).\mathit{dir}(\mathit{last}(p)) := \mathit{null}\ ]\ ) \\ \text{where } qq = L(\mathit{parent}(q),x) \text{ and } pp = L(\mathit{parent}(p),x) \end{array}$$

Note that $J2(x)$ implies that the file identifiers pp and qq are equal if and only if p and q have the same parent. If so, then $p \not\sqsubseteq q$ implies that $\mathit{last}(p)$ and $\mathit{last}(q)$ differ. The refinement theorem for *move* is:

Theorem 3.4 *Assume that* $J0(x) \ \wedge \ J1(x) \ \wedge \ J2(x)$*. Then we have* $\mathit{abstract}(\mathit{move}(p,q,x)) = \mathit{move}(p,q,\mathit{abstract}(x))$*.*

We have proved this with PVS (see [7]). The structure of the proof is the same as for *delete* and *create*. Due to the many case distinctions, it is cumbersome. We omit it because it is not illuminating.

3.4 Garbage collection

Unreachable nodes in the tree are useless. Garbage collection amounts to the removal of useless nodes. In the present context this is impossible because every store x is a total function. The best we can do is minimize the unreachable nodes. This is done as follows.

The set of reachable file identifiers is defined by

$$reach(x) = \{f \mid \exists\, p : L(p, x) = f\}\ .$$

As unreachable file identifiers are never inspected, we define *garbage collection* by

$$\begin{aligned} &gc : [StoreT \to StoreT]\ , \\ &gc(x)(f) = (f \in reach(x)\ ?\ x(f) : nodeT(\bot))\ . \end{aligned}$$

By a straightforward induction on the length of p, one proves that $L(p, gc(x)) = L(p, x)$ for all paths p. Having done this, one can easily prove that $abstract(gc(x)) = abstract(x)$. In words, garbage collection does not influence the meaning of the store.

3.5 Proofs of the invariants

It is straightforward to prove that the operations *write*, *delete*, *create*, *move*, and *gc* preserve the invariant $J0$, i.e., $J0(x)$ implies $J0(write(p, m, x))$ for all $x : StoreT$, and similarly for the other functions. The same is done for the invariant $J1$. Preservation of $J3$ under these five operations follows from the fact that they add at most one element (in the case of *create*) to the range of the store.

The invariant $J2$ uses function L, which is defined recursively. We therefore define two simpler invariants, which express that the file tree has no cycles and that all occurring file identifiers $\neq null$ are different:

$$\begin{aligned} &J2a(x) : \quad \forall\, f, n : x(f).dir(n) \neq rootId\ , \\ &J2b(x) : \quad \forall\, f, g, m, n : x(f).dir(m) = x(g).dir(n) \neq null \ \Rightarrow\ f = g \ \wedge\ m = n\ . \end{aligned}$$

Here, f and g range over *Fid* and m and n range over *Name*. By induction on the lengths of the paths, one proves that these two invariants, together with $J0$, imply $J2$. It is fairly easy to prove that *write*, *delete*, *move*, and *gc* preserve the invariants $J2a$ and $J2b$. For *create*, we use $J3$ and formula (1).

Finally, it is straightforward to prove that $initstoreT(d)$ satisfies the invariants $J0$, $J1$, $J2a$, $J2b$, and $J3$.

4 Implementing the Store

We now replace the total functions of the previous section by "finite maps", i.e., partial functions with a finite domain. We thus use the types declared in:

$$\begin{aligned} &DirI = [Name \to \texttt{lift}[Fid]]\ , \\ &NodeI = [\#\quad data : Data\ ,\ dir : DirI \quad \#]\ , \\ &StoreI = [Fid \to \texttt{lift}[Node]]\ . \end{aligned}$$

Working with partial functions in a theorem prover like PVS gives technical difficulties that, from a conceptual point of view, seem inessential and distracting. In the implementation, however, these difficulties correspond to the usual problems with unallocated pointers. It is therefore important to get it correct at the theoretical level.

In our presentation here, we make one simplification of the PVS code. If X is a type, the PVS type $\texttt{lift}[X]$ represents $X \cup \{\bot\}$, but X is not a subset of $\texttt{lift}[X]$. Instead, there is an injection $\texttt{up} : [X \to \texttt{lift}[X]]$ and an inverse coercion $\texttt{down} : [X' \to X]$ where $X' \subseteq \texttt{lift}[X]$ is the image of $\texttt{up}$. In the presentation below, we suppress the functions `up` and `down`, and regard X and X' as identical.

We construct a refinement function *refine* from the present system to the one of the previous section in:

$$\begin{aligned}
&\mathit{refine} : [\mathit{StoreI} \to \mathit{StoreT}] \ , \\
&\mathit{refine}(x)(f) = \\
&\quad (x(f) = \bot \ ? \ \mathit{nodeT}(\bot) \\
&\quad : (\#\ \mathit{data} := x(f).\mathit{data} \, , \mathit{dir} := \psi \circ (x(f).\mathit{dir})\ \#) \ ) \\
&\text{where } \psi(g) = (g = \bot \ ? \ \mathit{null} : g).
\end{aligned}$$

4.1 Reading and writing the store

The file identifier *null* is no longer needed in the implementation, but we allow and use it as an alias for $\bot$. We therefore define for $x : \mathit{StoreI}$ the invariant:

$K0(x): \quad x(\mathit{null}) = \bot \ .$

On the other hand, we want that all other file identifiers used in the store hold genuine nodes, as expressed in the invariant:

$$\begin{aligned}
K1(x): \quad & \forall\, f \in \mathit{range}(x) \ \Rightarrow \ f = \mathit{null} \ \vee \ x(f) \neq \bot \ , \text{ where} \\
& \mathit{range}(x) = \{\mathit{null}, \mathit{rootId}\} \cup \{f \in \mathit{Fid} \mid \exists\, g, n : f = x(g).\mathit{dir}(n)\} \ ,
\end{aligned}$$

where, by convention, $x(g).\mathit{dir}(n) \notin \mathit{Fid}$ when $x(g) = \bot$ or $x(g).\mathit{dir}(n) = \bot$.

At this refinement level, we use the *lookup* function L given by

$$\begin{aligned}
&L : [\mathit{StoreI} \times \mathit{Path} \to \mathit{Fid}] \ , \\
&L(p, x) = (\ p = \varepsilon \ ? \ \ \mathit{rootId} \\
&\qquad\qquad : \ x(L(\mathit{parent}(p), x)) = \bot \ \vee \\
&\qquad\qquad\quad\ x(L(\mathit{parent}(p), x)).\mathit{dir}(\mathit{last}(p)) = \bot \ ? \ \ \mathit{null} \\
&\qquad\qquad : \ x(L(\mathit{parent}(p), x)).\mathit{dir}(\mathit{last}(p)) \ ) \ .
\end{aligned}$$

The invariants $K0(x)$ and $K1(x)$ imply the rule:

$K01(x): \quad L(p, x) = \mathit{null} \quad \equiv \quad x(L(p, x)) = \bot \ .$

In PVS, reading store $x : \mathit{StoreI}$ at path p is defined by

$$\mathit{read}(p, x) = (x(L(p, x)) = \bot \ ? \ \bot \ : \ x(L(p, x)).\mathit{data}) \ .$$

A practical implementation would use the test $L(p, x) = \mathit{null}$ rather than the equivalent $x(L(p, x)) = \bot$. Doing this in PVS, however, would raise the objection that $x(L(p, x)).\mathit{data}$ is defined only if $x(L(p, x)) \neq \bot$. In other words, the function *read* would only be defined on the stores where $K01$ holds. Although we shall prove that $K01$ holds for all reachable stores, we prefer to define *read* as a total function in PVS and therefore use the definition above. The same argument applies to several of the definitions below.

Using a straightforward induction on the length of path p, one can prove

$$L(p, x) = L(p, \mathit{refine}(x)) \ .$$

This enables us to prove that $K01(x)$ implies $\mathit{read}(p, \mathit{refine}(x)) = \mathit{read}(p, x)$.

On this level, function *ls* is defined by

$$\mathit{ls}(p, x) = (x(L(p, x)) = \bot \ ? \ \bot : \mathit{ls}(x(L(p, x)).\mathit{dir})) \ , \text{ where}$$

$$\mathit{ls}(di) = \{n \in \mathit{Name} \mid di(n) \neq \bot \ \wedge \ di(n) \neq \mathit{null}\} \ .$$

Using the invariant $K0$, it is easy to prove the refinement theorem that $\mathit{ls}(p, \mathit{refine}(x)) = \mathit{ls}(p, x)$. Writing of store x is defined by

$$\begin{array}{l} \mathit{write}(p, m, x) = \\ \quad (\ x(L(p,x)) = \bot \ ?\ x \\ \quad :\ x \ \mathbf{with}\ [(L(p,x)).\mathit{data} := m(x(L(p,x)).\mathit{data})]\ )\ . \end{array}$$

Using $K01(x)$, one can prove that $\mathit{refine}(\mathit{write}(p, m, x)) = \mathit{write}(p, m, \mathit{refine}(x))$.

4.2 Tree modification

Analogously to the definition in section 3.1, here removal is defined by

$$\begin{array}{l} \mathit{delete}(p, x) = \\ \quad (\ p = \varepsilon \ \vee \ L(p,x) = \mathit{null}\ ?\ x \\ \quad :\ x \ \mathbf{with}\ [\ (pp).\mathit{dir}(\mathit{last}(p)) := \bot\ ]\ ) \\ \text{where} \ \ pp = L(\mathit{parent}(p), x). \end{array}$$

Note that in the second branch, $L(p, x) \neq \mathit{null}$ implies that $x(L(\mathit{parent}(p), x)) \neq \bot$. Therefore this node indeed has a directory that can be modified. The equality $\mathit{refine}(\mathit{delete}(p, x)) = \mathit{delete}(p, \mathit{refine}(x))$ is proved with the invariant $K01(x)$.

For making a directory, we again need finiteness of the store as expressed in the invariant

$$K2(x): \quad \#\mathit{range}(x) < \infty \ .$$

We can therefore define a function $\mathit{new} : [\mathit{Store} \to \mathit{Fid}]$ that satisfies $\mathit{new}(x) \notin \mathit{range}(x)$ for every x with $K2(x)$. We need a different node constructor (compare section 3):

$$\mathit{nodeI}(d) = (\#\ \mathit{data} := d\,,\ \mathit{dir} := (\lambda\, n : \bot)\ \#)\ .$$

Analogously to section 3.2, a new node is created by

$$\begin{array}{l} \mathit{create}(p, d, x) = \\ \quad (\ x(pp) = \bot \ \vee \ L(p,x) \neq \mathit{null}\ ?\ x \\ \quad :\ x \ \mathbf{with}\ [\ (\ pp\ ).\mathit{dir}(\mathit{last}(p)) := \mathit{ln}\,,\ (\mathit{ln}) := \mathit{node}(d)\ ]\ ) \\ \text{where} \ \ pp = L(\mathit{parent}(p), x) \text{ and } \mathit{ln} = \mathit{new}(x). \end{array}$$

It is easy to prove that $\mathit{range}(\mathit{refine}(x)) = \mathit{range}(x)$. We also get $\mathit{new}(\mathit{refine}(x)) = \mathit{new}(x)$, because we can use the same choice function. Using $K01(x)$, one can then prove the equality $\mathit{refine}(\mathit{create}(p, d, x)) = \mathit{create}(p, d, \mathit{refine}(x))$.

Function *move* is defined almost as in section 3.3:

$$\begin{array}{l} \mathit{move}(p, q, x) = \\ \quad (\ q = \varepsilon \ \vee \ p \sqsubseteq q \ \vee \ L(p,x) = \mathit{null} \ \vee \ x(qq) = \bot\ ?\ x \\ \quad :\ x \ \mathbf{with}\ [(qq).\mathit{dir}(\mathit{last}(q)) := L(p,x)\ , \\ \qquad\qquad\qquad (pp).\mathit{dir}(\mathit{last}(p)) := \bot\ ]\ ) \\ \text{where } qq = L(\mathit{parent}(q), x) \text{ and } pp = L(\mathit{parent}(p), x). \end{array}$$

At this point, the identification of type *Node* with a subtype of `lift`[*Node*] simplifies the presentation. Working in PVS, we need to make a case distinction whether the file identifiers *pp* and *qq* are equal or differ. Nevertheless, we formally proved the equality $\mathit{refine}(\mathit{move}(p,q,x)) = \mathit{move}(p,q,\mathit{refine}(x))$, using the invariant $K01$.

The verification that the invariants $K0$, $K1$, and $K2$ are preserved by the operations *write*, *delete*, *create*, and *move* are straightforward. These invariants also hold for the initial store defined by

$$\mathit{initstoreI}(d) = (\lambda\, f :\ f = \mathit{rootId}\,?\ \mathit{nodeI}(d) : \bot)\ .$$

Moreover, $\mathit{refine}(\mathit{initstoreI}(d)) = \mathit{initstoreT}(d)$.

It follows that the composition $\mathit{abs} = \mathit{abstract} \circ \mathit{refine}$ is a genuine refinement function $\mathit{Store} \to \mathit{StoreA}$.

4.3 Garbage and garbage collection

Garbage collection is more useful at this level than in section 3.4. Again we define:

$$\begin{aligned}&\mathit{reach} : [\mathit{StoreI} \to \mathbb{P}[\mathit{Fid}]]\ ,\\ &\mathit{reach}(x) = \{f \mid \exists\, p : L(p,x) = f\}\ .\end{aligned}$$

Garbage collection now means removal of unreachable nodes:

$$\begin{aligned}&\mathit{gc} : [\mathit{StoreI} \to \mathit{StoreI}]\ ,\\ &\mathit{gc}(x)(f) = (f \in \mathit{reach}(x)\ ?\ x(f) : \bot)\ .\end{aligned}$$

As before, one first proves that $L(p,\mathit{gc}(x)) = L(p,x)$ for all paths p and $x : \mathit{StoreI}$. Then it is, indeed, straightforward to prove that function gc preserves the three invariants $K0$, $K1$, and $K2$.

It is easy to prove that $\mathit{refine}(\mathit{gc}(x)) = \mathit{gc}(\mathit{refine}(x))$. It follows that the composition $\mathit{abs} : [\mathit{StoreI} \to \mathit{StoreA}]$ satisfies $\mathit{abs}(\mathit{gc}(x)) = \mathit{abs}(x)$ for all $x : \mathit{StoreI}$.

5 File Permissions at Three Levels

File system permissions form a core issue in every operating system. Not all users must be able to read and modify all data. We therefore overload the six file system functions by adding a user as a new first argument, where *User* is a new type, uninterpreted for now. For the sake of orthogonality, we deviate somewhat from the standard Unix conventions.

5.1 Permissions in the abstract system

We describe the file system permission model from the user's point of view at the abstract level. For the user, we have typical access types like reading, executing and writing, and the owner can control the permissions to these operations. Furthermore, there is the concept of a super user, who holds all access rights in the file system.

We assume that the permissions attached to a node are encoded in the data of the node by means of predicates:

$$\mathit{px}, \mathit{pr}, \mathit{pw} : \mathbb{P}[\mathit{User} \times \mathit{Data}]\ ,$$

where *px* stands for the permission to execute, *pr* to read, and *pw* to write. We do not go into details of how these permissions are represented in the data. Instead, we concentrate on the specification and verification that users can only access and modify according to the permissions granted. As the functions px, pr, pw depend on the user, they can also depend on the classification of the user as creator of the file or directory, as a member of the group, etc. We can therefore here ignore these issues. As we need to apply these predicates in stores at a given path, we overload them to

$$\begin{array}{lcl} px, pr, pw : \mathbb{P}[\mathit{User} \times \mathit{Path} \times \mathit{StoreA}] \ , \\ px(u, p, x) & = & x(p) \neq \bot \ \wedge \ px(u, x(p)) \ , \end{array}$$

and similarly for pr and pw.

In case of files, readable, executable and writable means that the contents of a file can be read, executed (if it is executable) and written. In case of directories, readable corresponds to the listing of the directory entries, and executable means that user is allowed to go into the directory, i.e., "change directory". Writable means the permission to create or remove entries in the given directory. Therefore, for reading and writing in a file or directory at some path, the user needs execution rights along the whole path in the file system [1, Section 2.8]. This implies that the effective permissions are slightly more complicated functions that depend on the user, the path, and the store. We thus define:

$$\begin{array}{lcl} pX, pR, pW : \mathbb{P}[\mathit{User} \times \mathit{Path} \times \mathit{StoreA}] \ , \\ pX(u, p, x) & = & (\forall\, q :\ q \sqsubseteq p \ \Rightarrow \ px(u, q, x) \ , \\ pR(u, p, x) & = & pr(u, p, x) \ \wedge \ (p = \varepsilon \ \vee \ pX(u, \mathit{parent}(p), x)) \ , \\ pW(u, p, x) & = & pw(u, p, x) \ \wedge \ (p = \varepsilon \ \vee \ pX(u, \mathit{parent}(p), x)) \ . \end{array}$$

Here, by convention, $\mathit{parent}(\varepsilon) = \varepsilon$. In some Unix variants, write permission may imply or require read permission. This can be modelled by adapting the relations of pw and pr to the actual permission bits.

The user-adapted abstract versions of ls, read, and write are simply:

$$\begin{array}{l} \mathit{ls}(u, p, x) = (pR(u, p, x) \ ? \ \mathit{ls}(p, x) \ : \ \bot) \ , \\ \mathit{read}(u, p, x) = (pR(u, p, x) \ ? \ x(p) \ : \ \bot) \ , \\ \mathit{write}(u, p, m, x) = (pW(u, p, x) \ ? \ \mathit{write}(p, m, x) \ : \ x) \ . \end{array}$$

For creation the path must be nonempty and the user needs permission to execute and write the parent directory. We therefore define

$$\begin{array}{l} pY(u, p, x) \ = \ pX(u, p, x) \ \wedge \ pw(u, p, x) \ , \\ \mathit{create}(u, p, d, x) = (pY(u, \mathit{parent}(p), x) \ ? \ \mathit{create}(p, d, x) \ : \ x) \ . \end{array}$$

For deletion (assuming the node holds a directory), we require that the directory at the node is empty and we need ls to verify this. We therefore define

$$\begin{array}{l} \mathit{delete}(u, p, x) = \\ \qquad (pW(u, \mathit{parent}(p), x) \ \wedge \ \mathit{ls}(u, p, x) = \emptyset \ ? \ \mathit{delete}(p, x) \ : \ x) \ . \end{array}$$

Note that the user u needs read permission to obtain $\mathit{ls}(u, p, x) = \emptyset$. Otherwise function ls yields $\bot$, and $\bot \neq \emptyset$.

For move, we propose:

$$\begin{array}{l} \mathit{move}(u, p, q, x) \ = \\ \quad (pY(u, \mathit{parent}(p), x) \ \wedge \ pW(u, \mathit{parent}(q), x) \ ? \ \mathit{move}(p, q, x) \ : \ x) \ . \end{array}$$

5.2 Refinement of permissions

We now turn from the abstract stores of section 2 to the total stores of section 3. We extend the permission bit functions *px*, *pr*, *pw* to the type `lift`[*Data*] by defining

$$px(u, \bot) = pr(u, \bot) = pw(u, \bot) = \mathit{false}\ .$$

The *lookup* function L that gives the file identifier of a path is now modified to verify execution permissions along the path:

$$\begin{aligned}
&L : [\mathit{User} \times \mathit{Path} \times \mathit{StoreT} \to \mathit{Fid}]\ ,\\
&L(u,p,x) = \\
&\quad (\ p = \varepsilon\ ?\ \ \mathit{rootId}\\
&\quad :\ px(u, \mathit{xpp.data})\ ?\ \mathit{xpp.dir}(\mathit{last}(p))\\
&\quad :\ \mathit{null}\ )\\
&\text{where}\ \ \mathit{xpp} = x(L(u, \mathit{parent}(p), x)).
\end{aligned}$$

This expresses that the user can only traverse a path p if he has rights to execute all strict ancestors of p. Indeed, under assumption of $J0(x)$ and $J1(x)$, we have

$$\begin{aligned}
&L(u,p,x) = \\
&\quad (p = \varepsilon\ \vee\ pX(u, \mathit{parent}(p), \mathit{abstract}(x))\ ?\ L(p,x) : \mathit{null})\ .
\end{aligned}$$

The proof of this is complicated. The result is at the basis of the theorems that the refinement function *abstract* respects (i.e., commutes with) the functions *read*, *ls*, *write*, *create*, *delete*, *move*, as defined below.

The user-adapted versions of *read* and *ls* are given by

$$\begin{aligned}
&\mathit{read}(u,p,x) = \\
&\quad (\ L(u,p,x) = \mathit{null}\ \vee\ \neg\, pr(u, x(L(u,p,x)).\mathit{data})\ ?\ \bot\\
&\quad :\ x(L(u,p,x)).\mathit{data})\ ,\\
&\mathit{ls}(u,p,x) = \\
&\quad (\ L(u,p,x) = \mathit{null}\ \vee\ \neg\, pr(u, x(L(u,p,x)).\mathit{data})\ ?\ \bot\\
&\quad :\ \mathit{ls}(x(L(u,p,x)).\mathit{dir})\ )\ .
\end{aligned}$$

The user-adapted version of *delete* becomes:

$$\begin{aligned}
&\mathit{delete}(u,p,x) = \\
&\quad (\ p = \varepsilon\ \vee\ \neg\, pw(u, x(pp).\mathit{data})\ \vee\ \mathit{ls}(u,p,x) \neq \emptyset\ ?\ x\\
&\quad :\ x\ \textbf{with}\ [\ (pp).\mathit{dir}(\mathit{last}(p)) := \mathit{null}\ ]\ )\\
&\text{where}\ \ pp = L(\mathit{parent}(p), x).
\end{aligned}$$

For the sake of brevity, we omit the definitions of *write*, *create*, and *move* at this level. Using the invariants $J0, \ldots, J3$, we then prove the refinement theorems for the user-adapted functions of this level, analogous to those of section 3. All details are given in the PVS proof script of [7].

5.3 Implementation of permissions

We now turn to the concrete stores of section 4. For the permission system, we extend the *lookup* function L of section 4 to verify the execution permissions along the path:

$$
\begin{aligned}
& L : [\mathit{User} \times \mathit{Path} \times \mathit{StoreI} \to \mathit{Fid}] \ , \\
& L(u,p,x) = (\ p = \varepsilon \ ? \ \ \mathit{rootId} \\
& \qquad\qquad\quad : \ x(L(u, \mathit{parent}(p), x)) = \bot \\
& \qquad\qquad\quad \vee \neg\, \mathit{px}(u, x(L(u, \mathit{parent}(p), x)).\mathit{data}) \\
& \qquad\qquad\quad \vee\, x(L(u, \mathit{parent}(p), x)).\mathit{dir}(\mathit{last}(p)) = \bot \ ? \ \ \mathit{null} \\
& \qquad\qquad\quad : \ x(L(u, \mathit{parent}(p), x)).\mathit{dir}(\mathit{last}(p))\ ) \ .
\end{aligned}
$$

The functions *read* and *ls* of section 4 are modified for the user-adapted version as:

$$
\begin{aligned}
& \mathit{read}(u,p,x) = \\
& \quad (\ x(L(u,p,x)) = \bot \ \vee \ \neg\, \mathit{pr}(u, x(L(u,p,x)).\mathit{data}) \ ? \ \bot \\
& \quad : \ x(L(u,p,x)).\mathit{data}) \ .
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{ls}(u,p,x) = \\
& \quad (\ x(L(u,p,x)) = \bot \ \vee \ \neg\, \mathit{pr}(u, x(L(u,p,x)).\mathit{data}) \ ? \ \bot \\
& \quad : \ \mathit{ls}(x(L(u,p,x)).\mathit{dir}) \ .
\end{aligned}
$$

The function *write* is modified analogously:

$$
\begin{aligned}
& \mathit{write}(p,m,x) = \\
& \quad (\ x(L(u,p,x)) = \bot \ \vee \ \neg\, \mathit{pw}(u, x(L(u,p,x)))\ ? \quad x \\
& \quad : \ x \ \mathbf{with} \ [(L(u,p,x)).\mathit{data} := m(x(L(u,p,x)).\mathit{data})]\ ) \ .
\end{aligned}
$$

Function *create* needs permissions for lookup, writing, and executing in the parent directory:

$$
\begin{aligned}
& \mathit{create}(u,p,d,x) = \\
& \quad (\ x(\mathit{pp}) = \bot \ \vee \ L(u,p,x) \neq \mathit{null} \\
& \quad \vee \neg\, \mathit{px}(u, x(\mathit{pp}).\mathit{data}) \ \vee \ \neg\, \mathit{pw}(u, x(\mathit{pp}).\mathit{data}) \ ? \quad x \\
& \quad \ : \ x \ \mathbf{with} \ [\ (\ \mathit{pp}\ ).\mathit{dir}(\mathit{last}(p)) := \mathit{ln}\,, \ (\mathit{ln}) := \mathit{node}(d) \ ]\ ) \\
& \text{where} \ \ \mathit{pp} = L(u, \mathit{parent}(p), x) \text{ and } \mathit{ln} = \mathit{new}(x).
\end{aligned}
$$

Function *delete* of section 4 becomes:

$$
\begin{aligned}
& \mathit{delete}(u,p,x) = \\
& \quad (\ p = \varepsilon \ \vee \ \mathit{ls}(u,p,x) \neq \emptyset \\
& \quad \vee\, x(\mathit{pp}) = \bot \ \vee \ \neg\, \mathit{pw}(u, x(\mathit{pp}).\mathit{data}) \ ? \ x \\
& \quad : \ x \ \mathbf{with} \ [\ (\mathit{pp}).\mathit{dir}(\mathit{last}(p)) := \bot\ ]\ ) \\
& \text{where} \ \ \mathit{pp} = L(\mathit{parent}(p), x).
\end{aligned}
$$

Here the condition $x(\mathit{pp}) \neq \bot$ is needed to read $x(\mathit{pp}).\mathit{data}$, because $\mathit{ls}(u,p,x) = \emptyset$ only implies $x(\mathit{pp}) \neq \bot$ under assumption of the invariant $K0(x)$.

We adapt function *move* of section 4 as:

$$
\begin{aligned}
& \mathit{move}(u,p,q,x) = \\
& \quad (\ q = \varepsilon \ \vee \ p \sqsubseteq q \ \vee \ L(u,p,x) = \mathit{null} \ \vee \ x(\mathit{qq}) = \bot \\
& \quad \vee \neg\, \mathit{pw}(u, x(\mathit{pp}).\mathit{data}) \ \vee \ \neg\, \mathit{pw}(u, x(\mathit{qq}).\mathit{data}) \ ? \ x \\
& \quad : \ x \ \mathbf{with} \ [(\mathit{qq}).\mathit{dir}(\mathit{last}(q)) := L(u,p,x) \ , \\
& \qquad\qquad\qquad (\mathit{pp}).\mathit{dir}(\mathit{last}(p)) := \bot\ ]\ ) \\
& \text{where } \mathit{qq} = L(u, \mathit{parent}(q), x) \text{ and } \mathit{pp} = L(u, \mathit{parent}(p), x)
\end{aligned}
$$

We finally prove with PVS, that the refinement function from the implemented store to the total store also respects (i.e., commutes with) the user-adapted versions of *read*, *write*, *ls*, *create*, *delete*, and *move*. The details of the proof can be found at

[7].

6 Conclusion

In this work, we constructed and proved the specifications of a hierarchical file system. We used functional refinements to model a file system, starting from an abstract version and working towards a concrete specification. We divided our work into four parts (i) Abstract model (ii) First refinement using total functions (iii) Final refinement using partial functions. Finally, (iv), at all three levels, we incorporated a permission mechanism like that of the UNIX file system.

Initially, we tried to model file systems directly at the implementation level of Section 4. In order to evade or at least postpone the details of partial functions, we invented the more abstract level of Section 3. The real breakthrough came when we saw that we had to begin by specifying a hierarchical file system from a user's point of view, as a partial function from (absolute) paths to data. The requirements for the other two levels then emerged naturally as proof obligations for the refinement functions. Having the three levels was also very helpful in the development of the permission system.

A total of 204 lemmas were proved with proof assistant PVS [6] during this work. It included 10 lemmas for the abstract model, 87 lemmas for the model with total functions, 79 lemmas for the model with partial functions, and 28 lemmas shared for all models. This may be an indication of the efficiency of PVS as compared to the work done in [17] using Athena where they constructed 283 lemmas and theorems for only reading and writing into files in only one directory. Details of the PVS proof can be found in the proof script for this work at [7].

As for directions for future research, the model needs an extension with hard links. At the abstract level, the appropriate way to do this may be by means of a modifiable equivalence relation on valid paths, as a second component of the store. Function *write* should then modify all members of the equivalence class of the path. A next extension could be to incorporate the difference between files and directories. After this, several problem areas ask for attention: the details of reading and writing, concurrent access, disk lay-out, distribution, and fault tolerance.

References

[1] Abrahams, P., Larson, B.: Unix for the Impatient. Addison-Wesley, Reading, etc. (1996)

[2] Yang, J., Twohey, P., Engler, D., Musuvathi, M.: Using model checking to find serious file system errors. ACM Transactions on Computer Systems **24** (2006) 393–423

[3] Huth, M., Ryan, M.: Logic in Computer Science: Modelling and reasoning about systems. 2nd edition edn. Cambridge University Press (2004)

[4] Lutz, R.: Analyzing software requirements errors in safety-critical embedded systems. In: IEEE International Symposium on Requirements Engineering, CA (1993) 126–133

[5] Pecheur, C.: Advanced modelling and verification techniques applied to a cluster file system. In: Proceedings of the 14th IEEE international conference on Automated software engineering, Washington, DC, USA, IEEE Computer Society (1999) 119–126

[6] Owre, S., Shankar, N., Rushby, J., Stringer-Calvert, D.: PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference. (2001) `http://pvs.csl.sri.com`

[7] Hesselink, W., Lali, M.: PVS proof script of "file system formalization". Available at: `www.cs.rug.nl/~wim/mechver/fs/index.html` (2009)

[8] Hoare, C.: The verifying compiler: A grand challenge for computing research. Journal of the ACM **50** (2003) 63–69

[9] Joshi, R., Holzmann, G.: A Mini Challenge: Build a verifiable filesystem. Formal Aspects of Computing **19** (2007) 4

[10] Woodcock, J., Banach, R.: The verification grand challenge. Computer Society of India Communications (2007) 661–668

[11] Morgan, C., Sufrin, B.: Specification of the UNIX filing System. IEEE Transactions on Software Engineering **SE-10** (1984) 128–142

[12] Freitas, L., Woodcock, J., Fu, Z.: POSIX file store in Z/Eves: An experiment in the verified software repository. Sci. Comput. Program. **74** (2009) 238–257

[13] Fu, Z.: A refinement of the UNIX filing system using Z/Eves. Master's thesis, University of York (2006)

[14] Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., G.T, Leino, K., Poll, E.: An overview of jml tools and applications. International Journal on Software Tools for Technology Transfer (2003) 73–89

[15] Wenzel, M.: Some aspects of Unix file-system security. Isabelle/Isar proof document. T.U. Munchen (2001)

[16] Galloway, A., Luttgen, G., Muhlberg, J., Siminiceanu, R.: Model-checking the Linux virtual file system. In: VMCAI. (2009) 74–88

[17] Arkoudas, K., Zee, K., Kuncak, V., Rinard, M.: Verifying a file system implementation. In: Sixth International Conference on Formal Engineering Methods (ICFEM04), volume 3308 of LNCS. (2004) 8–12

[18] Hughes, J.: Specifying a visual file system in Z. Technical report, Department of Computing Science, University of Glasgow, 3 pages (1989)

[19] Damchoom, K., Butler, M., Abrial, J.R.: Modelling and proof of a tree-structured file system in Event-B and Rodin. In: ICFEM. (2008) 25–44