Specifying a Realistic File System

Sidney Amani Toby Murray
NICTA and University of New South Wales, Australia

We present the most interesting elements of the correctness specification of BilbyFs, a performant Linux flash file system. The BilbyFs specification supports asynchronous writes, a feature that has been overlooked by several file system verification projects, and has been used to verify the correctness of BilbyFs's `fsync()` C implementation. It makes use of nondeterminism to be concise and is shallowly-embedded in higher-order logic.

1 Introduction

File systems have been a target of software verification research for decades [10, 1, 8]. While recent research has increased the fidelity of the specifications against which file system implementations are to be verified, the gap remains large between the details and features covered by existing verification work, on the one hand, and those for production file systems as found in e.g. the Linux kernel, on the other.

A common simplification has been to omit *asynchronous writes* from the file system specification and implementation [5, 6, 2]. File systems typically buffer state-updates in-memory, so that updates to the physical storage medium can be batched to improve throughput. Updates are propagated to the storage medium periodically, or explicitly via the file system `fsync()` operation, meaning that they occur asynchronously. This asynchrony is crucial for performance but complicates the file system specification and implementation.

In this paper, we describe the most interesting elements of the correctness specification of BilbyFs. BilbyFs is a custom flash file system we developed that runs as a Linux kernel module and supports asynchronous writes, and whose `fsync()` implementation has been verified as functionally correct against the correctness specification we present here. To make the verification tractable, BilbyFs enforces sequential execution of the file system code by acquiring a global lock before invoking an operation. While out of scope for this paper, BilbyFs performs comparably to mainstream flash file systems like UBIFS [7] and JFFS2 [14], and its design strikes a balance between the simplicity of JFFS2 and the runtime performance and reliability of UBIFS.

The BilbyFs specification is complementary to that of Chen et al. [2] who recently verified FSCQ, a crash-safe user-space file system implemented in Haskell. FSCQ performs asynchronous writes only *within* each individual file system operation, and synchronously waits for writes to complete at the end of each operation. In contrast, BilbyFs allows entire sequences of file system operations to occur asynchronously. Their specification describes FSCQ's FUSE interface, which is closely aligned with the POSIX interface expected by application programs, and is specified as a series of Hoare triples over the top-level functions. The BilbyFs specification is a functional program in Isabelle/HOL (§2) that describes the interface BilbyFs provides to the Linux kernel's Virtual File system Switch (VFS), which is at a different level of abstraction than FUSE. To support fully asynchronous operations, in contrast to Chen et al.'s, the BilbyFs specification explicitly separates the in-memory and on-medium file system state (§3) which allows us to specify the effect of the `fsync()` operation (§4) and reduce the gap between realistic file systems and verified ones (§5).

R.J. van Glabbeek, J.F. Groote & P. Höfner (Eds.): Workshop on Models for Formal Analysis of Real Systems (MARS 2015)
EPTCS 196, 2015, pp. 1–9, doi:10.4204/EPTCS.196.1

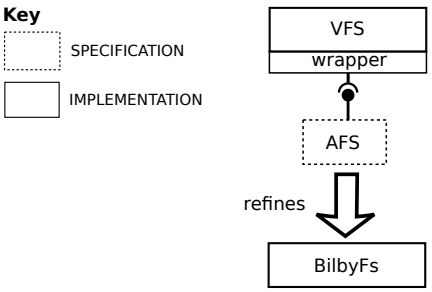

Figure 1: Correctness specification overview

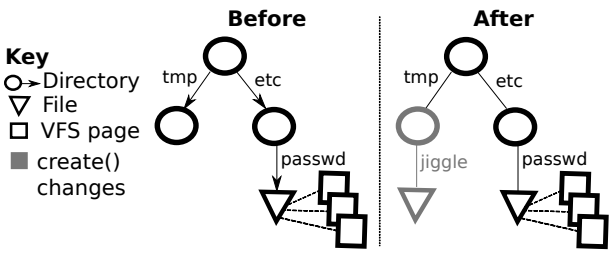

Figure 2: The effect of the `create()` operation

2 Formalisation

Our objective is to prove the *functional correctness* of BilbyFs, specifically that every behaviour exhibited by the implementation is captured by its specification (i.e. formal refinement [12]). The BilbyFs implementation uses a global lock to avoid all concurrency, meaning that its specification can be entirely sequential. This specification should be concise enough to be easily audited to ensure it captures the intended behaviour of BilbyFs.

We chose to shallowly-embed the correctness specification of BilbyFs in Isabelle/HOL [11]. Our specification is formalised in the *nondeterminism monad*, inspired by the nondeterministic state monad of Cock et al.[3], where computations are modelled as sets of values, one for each possible execution and associated return-value of the computation. A computation that returns results of type σ is modelled as a value of type σ *set*: singleton sets model deterministic computations; the empty set models computations whose semantics are not well-defined.

The primitive computation *return x* simply yields the result *x*: *return* $x \equiv \{x\}$. The "bind" operation, written $>>=$, sequences two computations together. $f >>= g$ is the specification that for each result r of the computation f, executes g passing r as g's argument: $f >>= g \equiv \bigcup_{r \in f} g\ r$. The nondeterministic choice $x \sqcap y$ between two computations x and y is defined trivially: $x \sqcap y \equiv x \cup y$. We write *do* $x \leftarrow f$; g *od* as sugar for $f >>= g$, in the style of Haskell.

The do-notation makes the specification readable by ordinary programmers. Being shallowly-embedded in Isabelle/HOL, the specification is more straightforward to reason about than if it were deeply-embedded. Finally, following [3], this formalism supports a scalable, compositional framework for proving refinement.

Unlike Cock et al., we eschew a state monad for the AFS. Our specifications are simple enough that passing the file system state explicitly adds little overhead. Further, we found that a state monad makes our specifications harder to read while imposing extra mental overhead due to having to unpack and repack the state when calling into sub-modules that only touch a small part of it.

3 File System Abstraction

Having described the formalism of our specification, we now describe more precisely the level of abstraction at which we specify file system behaviours and we present our abstract model of the file system state used to specify asynchronous writes.

Modern operating systems include a variety of file systems. Each in-kernel file system does not directly provide functionality to the user, but instead interacts via the kernel's VFS that provides a common

interface to all file systems. Thus the top-level operations provided by BilbyFs, and described by its correctness specification, are those expected by the VFS. These include a total of 16 file system operations. `create()`, `unlink()`, `mkdir()` and `rmdir()` for respectively creating and removing both files and directories, as well as `readpage()`, `write_begin()`, `write_end()`, `readdir()` for reading and writing files and listing directories. `lookup()` finds an inode number by name and reads it from disk. `rename()` renames or move a file or directory. `symlink()` creates a symbolic link and `follow_link()` reads the path stored in a symbolic link. `link()` creates a hardlink to a file. `setattr()` and `getattr()` for accessing and altering attributes of file and directories. Finally, `evict_inode()` is called when an file is evicted from the cache and `fsync()` synchronises in-memory updates to storage.

Figure 1 shows how the correctness specification, which we call the *Abstract File system Specification* (AFS), relates to the Linux VFS and to the BilbyFs implementation. A small C wrapper that sits between the Linux VFS and BilbyFs decouples VFS interfaces from the Linux kernel caches and serialises the execution of file system operations (see Section 5). BilbyFs operations are invoked by the VFS through the wrapper; the AFS describes concisely how these operations should behave.

The BilbyFs functional correctness proof, completed for the `fsync()` operation, establishes that the BilbyFs implementation correctly implements the AFS, by proving that the former formally *refines* [12] the latter.

Modelling the File System State Internally, a file system implements its hierarchical structure on disk in linked objects called *inodes*, which are file system specific. Similarly to previously-mentioned work [5, 6], the AFS models the file system state as the type *afs-map*, which is a synonym for a (partial) mapping from *inode numbers* (32-bit integers that uniquely identify inodes) to *afs-inode* objects. *afs-inode*s abstractly model the on-medium representation of BilbyFs inodes.

type-synonym *afs-map* = *ino* ⇀ *afs-inode*

datatype *afs-inode-type* =
IDir filename ⇀ *ino*
| *IFile vfspage list*

record *afs-inode* =
i-type :: *afs-inode-type*
i-ino :: *ino*
i-nlink :: *32 word*
i-size :: *64 word*
...

An *afs-inode* represents either a directory or a file, and the *i-type* field stores both what kind of object it represents as well as the content it holds in each case. Each directory inode holds a mapping from filenames (byte strings) to associated inode numbers. Each file inode links to the data pages that hold its file's contents.

This abstraction allows the AFS to be as simple as possible while comprehensively capturing the behaviour of the various file system operations. In practice, inode implementations are a source of complexity and defects [9] for file systems. For instance the inode contents described above, which the AFS models as stored directly in the *i-type* field of the *afs-inode* object, will in practice be stored in separate data structures on the medium to which the inode links, often via multiple levels of indirection.

The AFS represents a file's contents as a list of VFS pages (each of which is a list of bytes). An alternative would have been to use just a single flat list of bytes. However, we chose the former because the VFS

layer interacts closely with the Linux page cache (i.e. memory management subsystem) which is directly reflected in the VFS API that BilbyFs implements.

Directory inodes are usually implemented as a collection of directory entries stored in a file. The AFS instead models directory contents as a mapping from filenames to inode identifiers: the inode identifier for a particular filename identifies the file inode that implements the file with that name. In file systems like BilbyFs that support *hardlinks*, the same file inode may be linked to by multiple filenames (in possibly different directories). Like most file systems, BilbyFs restricts hardlinks to point only to file inodes, to ensure that the file system's logical structure is acyclic.

AFS Invariant The aforementioned requirement on hardlinks is encoded as part of the global invariant of the AFS, which we describe only briefly because (like the AFS state itself) it is very similar to that of [4]. The invariant assertion includes the following requirements of the *afs-map* structure. (1) A root inode exists; (2) for each directory inode, there is only one directory with an entry pointing to that inode — i.e. no hardlinks on directories; (3) the *i-ino* field of each inode matches the *afs-map* mapping; (4) the *i-nlink* field of each inode correctly tracks the number of links to that inode; and (5) the *i-size* field of each file inode is consistent with the number of VFS pages attached to the inode.

Modelling File System Updates The AFS's abstract representation of the file system state makes describing the effects of the file system operations easy, and in turn facilitates auditing of the specification.

Figure 2 depicts the changes to the *afs-map* structure when the `create()` operation is called to create a new file `jiggle` in the `/tmp` directory. Grey nodes are those altered or added by the operation. Creating a file involves adding a file inode and a link to it in the parent directory. Directories are pictured as circles with arrows denoting each entry in the directory. Small triangles denote files and are linked to VFS pages shown as tiny squares. The newly created file contains no data so no VFS page is attached to its inode.

The effect on the *afs-map* structure m of creating a new file $''jiggle''$ in a directory (whose inode is) *itmp*, by installing the new file inode *ijiggle*, is specified by the following single line of Isabelle/HOL[1].

$$m\langle i\text{-}ino\ itmp \mapsto i\text{-}dir\text{-}upd\ (\lambda dir.\ dir\langle ''jiggle'' \mapsto i\text{-}ino\ ijiggle\rangle)\ itmp,\ i\text{-}ino\ ijiggle \mapsto ijiggle\rangle$$

We write $m\langle a \mapsto b\rangle$ to denote updating the (partial) map m such that a maps to b.

Each transformation to the file system state may be captured by a function of type $afs\text{-}map \Rightarrow afs\text{-}map$. We call such functions *file system updates*. We exploit this idea in the following section, where we describe the model of asynchronous writes used in the AFS.

Specifying Asynchronous Writes The *afs-map* type models the state of the file system as stored on the physical storage medium, which in the case of BilbyFs is the raw flash storage device, and updates to the storage medium are simply transformations: $afs\text{-}map \Rightarrow afs\text{-}map$. Like many other realistic file systems, although the effects of file system operations like `create()`, `unlink()` etc., become visible as soon as those invocations return, the actual storage medium update that they implement may not be applied until some point in the future, for instance when `fsync()` is next invoked. Thus writes to the storage medium are performed *asynchronously*, an essential feature of file systems since the original UNIX [13]. Storage medium updates are therefore buffered in-memory, allowing operations like `create()` to return

[1] Note that the specification for the `create()` operation (see Figure 3) is a lot more complex than this single line, because it needs to also incorporate error checking and handling, and interaction with the Linux VFS.

straightaway, without incurring the cost writing to the storage medium. For the file system correctness specification, this implies that the in-memory file system state and the state of the physical storage medium (*afs-map* in the AFS) need to be distinguished, especially if the semantics of operations like `fsync()` are to be precisely specified.

As mentioned in Section 1, several file system models [5, 6, 2] overlook this requirement. In the AFS for BilbyFs, the pending writes buffered in-memory are modelled as a sequence of file system transformations, each of type $afs\text{-}map \Rightarrow afs\text{-}map$. The global state of the specification is captured by the type *afs-state*. Besides the state of the physical medium, of type *afs-map*; and in-memory pending updates, of type $(afs\text{-}map \Rightarrow afs\text{-}map)\ list$; *afs-state* also includes a boolean flag that tracks whether the file system has been placed into *read-only mode*, which can occur for instance if a disk failure arises; as well as a record of the current time, which is used for instance when updating the timestamps on an inode that track e.g. the last time it was modified.

record *afs-state* =
  a-is-readonly :: *bool*
  a-current-time :: *time*
  a-medium-afs :: *afs-map*
  a-medium-updates :: $(afs\text{-}map \Rightarrow afs\text{-}map)\ list$

To model the idea that file system modifications become visible straightaway, even when they have not yet been applied to the physical storage medium, the AFS needs a way to calculate the (hypothetical) file system state that includes both the physical medium and the pending in-memory updates, i.e. the state that *would* arise if all of those updates were applied to the medium. It is this hypothetical state that must be considered, for instance, when the `unlink()` operation is invoked to remove an inode that was previously `create()`ed but hasn't yet been `fsync()`ed to disk. It is calculated by the function *updated-afs*, that makes use of the standard *fold* function to apply the in-memory updates to the medium state:

updated-afs afs-state ≡
fold $(\lambda x.\ x)$ (*a-medium-updates afs-state*) (*a-medium-afs afs-state*)

An operation like `create()` that updates the file system state may buffer the updates it performs in memory, or (if the in-memory buffer is full) it may cause preceding updates to be applied to the storage medium. Given that the size of the in-memory buffer is below the level of abstraction of the AFS, the precise number of updates that may be propagated to the storage medium could vary upwards from zero. The AFS captures this effect via the following helper function, which nondeterministically splits the list of updates into two parts: *to-apply*, the updates to be applied to the medium; and *rem*, the remainder. It then applies the updates in *to-apply* and updates the in-memory list of pending updates to *rem*.

afs-apply-updates-nondet afs ≡
do $(to\text{-}apply, rem) \leftarrow \{(t, r) \mid t\ @\ r = a\text{-}medium\text{-}updates\ afs\}$;
  return
   (*afs*(|*a-medium-afs* := *fold* $(\lambda x.\ x)$ *to-apply* (*a-medium-afs afs*),
        a-medium-updates := *rem*|))
od

The following helper function *afs-update* then generically specifies the process for updating the file system state, and is used in the specifications of the various file system operations (see e.g. `create()` in Figure 3). It takes an update function *upd* of type $afs\text{-}map \Rightarrow afs\text{-}map$. It adds it to the back of the list of in-memory updates and then calls *afs-apply-updates-nondet*. If after *afs-apply-updates-nondet* returns, the list of

in-memory updates is empty, then *afs-apply-updates-nondet* caused all in-memory updates (including *upd*) to be propagated to the storage medium, in which case the modification must report *Success*. Otherwise, it might succeed (if, for instance, the new update is simply buffered in-memory without touching the storage medium), or report an appropriate error (because a write to the medium failed, or a memory allocation error occurred etc.). If an error is reported, the new update *upd* is forgotten, ensuring operations that report an error do not modify the (combined in-memory and on-medium) file system state.

```
afs-update afs upd ≡
do afs ←
    afs-apply-updates-nondet
     (afs(|a-medium-updates := a-medium-updates afs @ [upd]|));
  if a-medium-updates afs = [] then return (afs, Success ())
  else return (afs, Success ()) ⊓
       nondet-error {eIO, eNoSpc, eNoMem}
        (λe. (afs(|a-medium-updates := butlast (a-medium-updates afs)|),
              Error e))
od
```

Importantly, this specification requires that no updates get lost when an error occurs: each is either applied (in order), or is still in the list of pending updates (in order). It also requires the BilbyFs implementation not to report an error if it succeeds in propagating all updates to disk. Thus the implementation cannot attempt to allocate memory, for instance, after successfully writing to disk. In practice, file system implementations structure each operation such that all resource allocation (and other actions that could potentially fail) occur early, so no potentially-failing operation needs to be performed after successfully updating the storage medium.

The *afs-update* definition is the heart of how the AFS specifies asynchronous file system operations while keeping the AFS concise and readable. In the following section, we present specifications of the most interesting top-level file system operations from the AFS.

4 Specifying File System Operations

Specifying `create()` The specification for the `create()` operation is shown in Figure 3. Recall that BilbyFs's top-level operations, like `create()`, are those expected by the Linux VFS. Since the VFS interacts with a range of different file systems, each of which may have its own custom inode format, the VFS provides a common inode abstraction, called a *vnode*. Top-level file system operations invoked by the VFS often take vnodes as their arguments and return updated vnodes in their results. The *afs-inode* structure mentioned in Section 3 is very similar to the generic vnode structure of the VFS.

Much of the complexity of Figure 3 comes from error checking and handling, as well as specifying the correct interaction with the VFS (e.g. conversion from vnodes to *afs-inode*s). The state of the file system, of type *afs-state*, is passed as the argument *afs*. `create()` also takes the parent directory vnode *vdir*, the name *name* of the file to create, a mode attribute *mode*, and a vnode *vnode* to fill with the information of the newly created file. It returns three values: the updated file system state, the updated parent directory vnode, and the updated vnode.

The specification precisely describes the file system behaviour expected by the VFS, including possible failure modes. For instance, the implementation needs to return an error if the file system is in read-only mode (line 1). On line 2, it allocates a new inode number and initialises the vnode fields by calling the

```
afs-create afs vdir name mode vnode ≡
1    if a-is-readonly afs then return ((afs, vdir, vnode), Error eRoFs)
2    else do r ← afs-init-inode afs vdir vnode (mode || s-IFREG);
3         case r of Error (afs, vnode) ⇒ return ((afs, vdir, vnode), Error eNFile)
4         | Success (afs, vnode) ⇒
5           do r ← read-afs-inode afs (v-ino vdir);
6             case r of Error e ⇒ return ((afs, vdir, vnode), Error e)
7             | Success dir ⇒
8               do r ← return (Success (i-dir-update (λd. d⟨αwa name ↦ v-ino vnode⟩) dir)) ⊓
9                       return (Error eNameTooLong);
10                case r of Error e ⇒ return ((afs, vdir, vnode), Error e)
11                | Success dir ⇒
12                  do r ← Success ‘ {sz | v-size vdir < sz} ⊓ return (Error eOverflow);
13                    case r of Error e ⇒ return ((afs, vdir, vnode), Error e)
14                    | Success newsz ⇒
15                      do time ← return (v-ctime vnode);
16                        dir ← return (dir(|i-ctime := time, i-mtime := time|));
17                        inode ← return (afs-inode-from-vnode vnode);
18                        (afs, r) ← afs-update afs (λf. f⟨i-ino inode ↦ inode, i-ino dir ↦ dir⟩);
19                      case r of Error e ⇒ return ((afs, vdir, vnode), Error e)
20                      | Success () ⇒
21                        return ((afs, vdir(|v-ctime := time, v-mtime := time, v-size := newsz|), vnode),
22                                Success ())
23                      od
24                  od
25              od
26          od
27   od
```

Figure 3: Functional specification of the *create* operation.

function *afs-init-inode* (not shown). If *afs-init-inode* returns an *Error* (line 3), `create()` returns *afs*, *vdir* and *vnode* unchanged as well as the error *Error eNFile* indicating the file system ran out of inode numbers. This pattern is repeated on lines 6, 10, 13 and 19, which each check for errors in preceding operations and specify that these errors must be propagated to `create()`'s caller, leaving the file system state unchanged.

On line 5, the specification reads the parent directory given by the *vnode* argument, and computes what the new directory should look like (with the file being created added to it). Line 12 implies that the size of the directory must increase. The core of `create()` is specified on line 18 where the file system state is updated with the updated directory and file inode.

Specifying `fsync()` Figure 4 shows the specification for `fsync()`, the file system operation that propagates all in-memory updates to the disk. `fsync()` returns an appropriate error when the file system is in read-only mode. Otherwise, it propagates the in-memory updates to the medium using the *afs-apply-updates-nondet* function of Section 2. Recall that this function applies the first n in-memory updates, with n chosen nondeterministically to model the effect of e.g. a disk failure happening part-way through. `fsync()` returns successfully when all updates are applied; otherwise it returns an appropriate error code. When an I/O error occurs (*eIO*), indicating a storage medium failure, the file system is put into read-only mode to prevent any further updates to the medium (whose state may now be inconsistent).

```
afs-fsync afs ≡
1   if a-is-readonly afs then return (afs, Error eRoFs)
2   else do afs ← afs-apply-updates-nondet afs;
3           if a-medium-updates afs = [] then return (afs, Success ())
4           else do e ← select {eIO, eNoMem, eNoSpc, eOverflow};
5                   return (afs(|a-is-readonly := e = eIO|), Error e)
6                od
7        od
```

Figure 4: Functional specification of the *fsync* operation.

The economy of the `fsync()` specification shows the advantage we obtain by carefully choosing an appropriate representation for the in-memory updates, separate from the on disk state of the file system.

5 Limitations

We conclude with a discussion of the BilbyFs AFS in order to tease out its limitations and avenues for future improvement. An obvious limitation of the AFS is that, like all other file systems specifications of which we are aware, it supports no form of concurrency, and so implicitly specifies that top-level operations cannot run concurrently to one another.

Another limitation of the AFS is that it imposes a strict ordering on all updates to be applied to the storage medium. In practice, many other file systems impose weaker ordering constraints, especially file systems that are highly concurrent or those built on top of low-level block interfaces that are asynchronous. Some file systems can reorder asynchronous writes of data but not those for meta-data. Investigating how to specify these weaker guarantees while still retaining the economy and simplicity of the AFS is an obvious avenue for future work.

The AFS, besides excluding concurrency, also does not specify the interaction between BilbyFs and the Linux kernel's inode, directory entry and page caches. The BilbyFs implementation makes use of a small C wrapper, pictured in Figure 1, that sits between it and the Linux VFS, which manages these caches and implements a global locking discipline that ensures that no two invocations of the BilbyFs file system operations can ever run concurrently to each other. This ensures that the AFS need not concern itself with the aforementioned Linux caches; however, specifying the behaviour of the Linux caching layers and their correct interaction with BilbyFs might be another interesting area for future work.

Another limitation of the AFS arises because function arguments are passed by value (rather than e.g. as pointers to variables in a mutable heap). This prevents the AFS from specifying VFS operations that take multiple pointer arguments that point to the same variable (i.e. are aliases for each other). Fortunately, the only top-level function able to take arguments that may alias is the VFS `rename()` operation, which takes two directory pointer arguments that respectively identify the source and target directory of the file to be renamed. When a file is renamed without changing its directory, the two directory arguments will alias. We exclude all such aliasing from BilbyFs by having the C wrapper check for this case: when the two pointers alias it invokes a separate top-level function of BilbyFs `rename()` that is a special case of the `move()` operation for when the source and target directory are identical.

A final limitation of the AFS presented here is that, unlike e.g. the recent work of Chen et al. [2], it does not specify the correct behaviour of the `mount()` operation, which is called at boot time, nor specify the file system state following a crash, for instance to require that the file system is crash-tolerant.

References

[1] William Bevier, Richard Cohen & Jeff Turner (1995): *A Specification for the Synergy File System*. Technical Report Technical Report 120, Computational Logic Inc., Austin, Texas, USA. Available at http://computationallogic.com/reports/files/120.pdf.

[2] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek & Nickolai Zeldovich (2015): *Using Crash Hoare Logic for Certifying the FSCQ File System*. Monterey, CA, doi:10.1145/2815400.2815402. To appear.

[3] David Cock, Gerwin Klein & Thomas Sewell (2008): *Secure Microkernels, State Monads and Scalable Refinement*. In: *21st TPHOLs*, Montreal, Canada, pp. 167–182, doi:10.1007/978-3-540-71067-7_16.

[4] Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler & Wolfgang Reif (2012): *A formal model of a virtual filesystem switch*. *arXiv preprint arXiv:1211.6187*, doi:10.4204/EPTCS.102.5.

[5] Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler & Wolfgang Reif (2013): *Verification of a Virtual Filesystem Switch*. In: *VSTTE 2013*, *LNCS* 8164, Menlo Park, CA, USA, pp. 242–261, doi:10.1007/978-3-642-54108-7_13.

[6] Wim H. Hesselink & Muhammad Ikram Lali (2009): *Formalizing a Hierarchical File System*. In: *14th REFINE*, *ENTCS* 259, Eindhoven, The Netherlands, pp. 67–85, doi:10.1016/j.entcs.2009.12.018.

[7] Adrian Hunter (2008): *A brief introduction to the design of UBIFS*.

[8] Rajeev Joshi & Gerard J Holzmann (2007): *A mini challenge: build a verifiable filesystem*. *Formal Aspects of Computing* 19(2), pp. 269–272, doi:10.1007/s00165-006-0022-3.

[9] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau & Shan Lu (2014): *A Study of Linux File System Evolution* 10(1), pp. 3:1–3:32. doi:10.1145/2560012.

[10] Carroll Morgan & Bernard Sufrin (1984): *Specification of the UNIX Filing System*. *Trans. Softw. Engin.* 10(2), pp. 128–142, doi:10.1109/TSE.1984.5010215.

[11] Tobias Nipkow, Lawrence Paulson & Markus Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, doi:10.1007/3-540-45949-9.

[12] Willem-Paul de Roever & Kai Engelhardt (1998): *Data Refinement: Model-Oriented Proof Methods and their Comparison*. 47, United Kingdom, doi:10.1017/CBO9780511663079.

[13] Ken Thompson (1978): *UNIX Time-Sharing System: UNIX Implementation*. *Bell System Technical Journal* 57(6), pp. 1931–1946, doi:10.1002/j.1538-7305.1978.tb02137.x.

[14] David Woodhouse (2003): *Jffs2: The journalling flash file system, version 2*. In: *proceedings of Ottawa Linux Symposium*.