Paper accepted for FM 2009: Formal Methods © 2009 Springer-Verlag Berlin Heidelberg
The final authenticated version is available online at https://doi.org/10.1007/978-3-642-05089-3_13

Abstract Specification of the UBIFS File System for Flash Memory

Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg, Germany
{schierl,schellhorn,haneberg,reif}@informatik.uni-augsburg.de

Abstract. Today we see an increasing demand for flash memory because it has certain advantages like resistance against kinetic shock. However, reliable data storage also requires a specialized file system knowing and handling the limitations of flash memory. This paper develops a formal, abstract model for the UBIFS flash file system, which has recently been included in the Linux kernel. We develop formal specifications for the core components of the file system: the inode-based file store, the flash index, its cached copy in the RAM and the journal to save the differences. Based on these data structures we give an abstract specification of the interface operations of UBIFS and prove some of the most important properties using the interactive verification system KIV.

1 Introduction

Flash memory has become popular in recent years as a robust medium to store data. Its main advantage compared to traditional hard disks is that it has no moving parts and is therefore much less susceptible to mechanical shocks or vibration. Therefore it is popular in digital audio players, digital cameras and mobile phones.

Flash memory is also getting more and more important in embedded systems (e.g. automotive [28]) where space restrictions rule out magnetic drives, as well as in mass storage systems (solid state disk storage systems like the RamSan-5000 from Texas Memory Systems) since it has shorter access times than hard disks.

Flash memory has different characteristics when compared to a traditional hard disk. These are explained in Section 2. In brief, flash memory cannot be overwritten, but only erased in blocks and erasing should be done evenly ("wear leveling"). These properties imply that standard file systems cannot be used with flash memory directly.

Two solutions are possible: either a flash translation layer is implemented (typically in hardware), translating standard file system operations into flash operations. This is the standard solution used e.g. in USB flash drives. It has the advantage that any file system can be used on top (e.g. NTFS or ext2). On the other hand, the characteristics of file systems (e.g. partitioning of the data

into the content of files, directory trees, or other meta data like journals etc.) cannot be effectively exploited using this solution.

Therefore a number of flash file systems (abbreviated FFS in the following) has been developed, that optimize the file system structure to be used with flash memory. Many of these FFS are proprietary (see [9] for an overview). A very recent development is UBIFS [14], which was added to the Linux kernel last year.

Increased use of flash memory in safety-critical applications has led Joshi and Holzmann [16] from the NASA Jet Propulsion Laboratory in 2007 to propose the verification of a FFS as a new challenge in Hoare's verification Grand Challenge [13]. Their goal was a verified FFS for use in future missions. NASA already uses flash memory in spacecraft, e.g. on the Mars Exploration Rovers. This already had nearly disastrous consequences as the Mars Rover Spirit was almost lost due to an anomaly in the software access to the flash store [22].

A roadmap to solving the challenge has been published by Freitas, Woodcock and Butterfield [8]. This paper presents our first steps towards solving this challenge.

There has been other work on the formal specification of file systems. First, some papers exist which start top-down by specifying directory trees and POSIX like file operations, e.g. the specifications of [20] of a UNIX-like file system, or our own specification of a mandatory security model for file systems on smart cards [26]. More recently and targeted towards the Grand Challenge Damchoom, Butler and Abrial [5] have given a high-level model of directory trees and some refinements. An abstract specification of POSIX is also given in [7] (some results are also in [21]). Butterfield and Woodcock [4] have started bottom-up with a formal specification of the ONFI standard of flash memory itself. The most elaborate work we are aware of is the one by Kang and Jackson [17] using Alloy. Its relation to our work will be discussed in Section 7. Our approach is middle-out, since our main goal was to understand the critical requirements of an efficient, real implementation. Therefore we have analyzed the code of UBIFS (ca. 35.000 loc), and developed an abstract, formal model from it. Although the resulting model is still *very* abstract and leaves out a lot of relevant details, it already covers some of the important aspects of any FFS implementation. These are:

1. Updates on flash are out-of-place because overwriting is impossible.
2. Like most traditional file systems the FFS is structured as a graph of inodes.
3. For efficiency, the main index data structure is cached in RAM.
4. Due to e.g. a system crash the RAM index can always get lost. The FFS stores a *journal* to recover from such a crash with a *replay* operation.
5. Care has been taken that the elementary assignments in the file system operations will map to atomic updates in the final implementation, to ensure that all intermediate states will be recoverable.

The paper is organized as follows. Section 2 gives an overview over the data structures and how the implementation works. Section 3 gives details of the

structure of inodes, and how they represent directory trees. The formal specifications we have set up in our interactive theorem prover KIV are explained. Section 4 explains how the journal works. Section 5 lists the specified file system operations and gives the code specifying the 'create file' operation as an example. Section 6 lists the properties we have verified with KIV and discusses the effort needed. Full specifications and proofs are available from the Web [18].

Section 7 presents a number of topics which still require future work to bring our abstract model close to a verified implementation. In particular, our model, just as UBIFS, does not consider wear leveling, but relegates it to a separate, lower level, called UBI ("unsorted block images"). Finally, Section 8 concludes.

2 Flash memory and UBIFS

Flash memory has certain special characteristics that require a treatment that is substantially different from magnetic storage. The crucial difference is that in-place updates of data, i.e. overwriting stored data, are not possible. To write new data on a currently used part of the flash memory, that part must first be erased, i.e. set back to an unwritten state. Flash media are organized in so-called erase blocks, which are the smallest data units that can be erased (typically 64 KB). Simulating in-place updates by reading a complete erase block, resetting it and writing back the modified erase block is not viable for two reasons. First, it is about 100 times slower than writing the modified data to a free erase block and second, it wears out the media. This is due to the second great difference between magnetic and flash storage. Flash memory gets destroyed by erasing. Depending on the used flash technology, the flash storage is destroyed after 10.000 to 2.000.000 erase cycles. This requires special FFS treatment, because the FFS must deal with the problem of deterioration of parts of the media that are erased often and with the fact that in-place changes of already written data are not possible. Therefore data is updated out-of-place instead, i.e. the new version of the data is written somewhere else on the media. This entails the need for *garbage collection* because sooner or later the different erase blocks all contain parts with valid data and parts with obsolete data. Garbage collection must be able to efficiently decide if an entry in an erase block still belongs to a valid file or directory. This is done by storing additional meta-data with the actual data. The combination of metadata and data is called a *node*.

A node records which file (or to be precise which *inode*) the node belongs to, what kind of data is stored in the node and the data themselves. The structure of the nodes in UBIFS and our resulting specification are described in Sect. 3. In our model, the nodes that are stored on the flash are contained in the flash store. The flash store is modeled as a finite mapping of *addresses* to nodes. Figure 1 shows the 4 central data structures of UBIFS (one of them the flash store) and explains what impacts the different operations have on them. The flash store is represented by the third column in Fig. 1, initially containing some data *FS* and some empty areas ($\emptyset$). In step ① of the figure, a regular operation (*OP*) is performed, e.g. overwriting data in a file. The second line shows the new state:

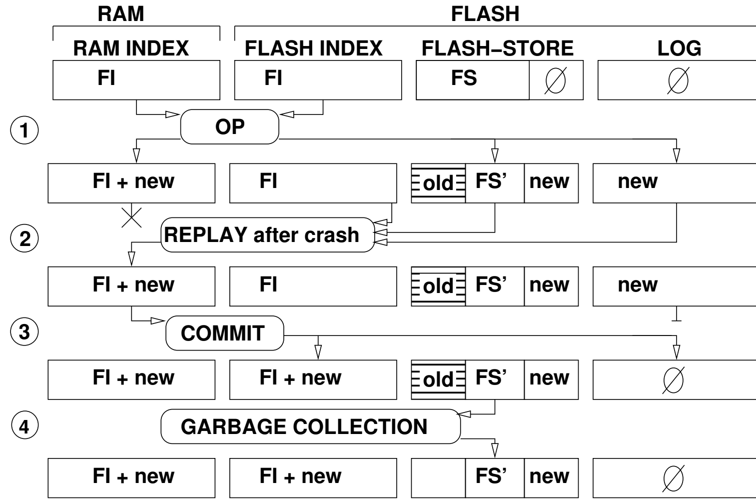

Fig. 1. Impact of UBIFS operations on the data structures

The flash store has some new data (*new*), some unchanged parts (FS') and some obsolete data (*old*) due to the out-of-place updates. These obsolete data can be collected by garbage collection as shown in step (4).

A problem that is crucial for the efficiency of the FFS is indexing. Without an index that allows searching for a specific node efficiently, the whole media would have to be scanned. Therefore UBIFS uses an index that maps so-called *keys* (which are, roughly speaking, unique identifiers of nodes) to the address of the node on the flash media. The index in UBIFS is organized as a B^+-tree and stored in memory (later on called *RAM index*). For efficiency reasons, the index should be stored on flash as well, because rebuilding the index by scanning the complete media (as JFFS2 [29] does) takes up a lot of time at mount time. Therefore UBIFS also stores the index on flash (later on *flash index*). The RAM index and the flash index are shown as the two leftmost columns in Fig. 1. Having an index on flash poses some difficulties as

– the index must be updated out-of-place on the flash media (this is done by using a wandering tree).
– the index has to be changed every time old data is changed or new data is written (because the new data must be added to the index and the position of the modified data changes due to out-of-place updates).

To limit the number of changes to the flash index, UBIFS does not update the flash index immediately, but uses a *journal* (also called *log*) of recent changes instead. Section 4 gives details on our model of the UBIFS journal. The log can be seen in the rightmost column of Fig. 1. Its use is illustrated in the second line of Fig. 1: The log contains references to the new data written to the flash store

in step (1) and the RAM index was updated accordingly ($FI + new$) while the flash index remained unchanged.

At certain points in time, the flash index is synchronized with the RAM index by the *commit* operation. This can be seen as step (3) in Fig. 1: The flash index is replaced with the current RAM index and the log is emptied[1]. One problem remains: What happens when the system crashes (e.g. power-failure) before the RAM index is written to flash? In this case, the flash index is out-of-date, compared to the data in the flash store. This problem is solved by the journal, because it records all changes to the data on flash that have not yet been added to the flash index. A special operation, *replay*, is done after a system crash to recover an up-to-date RAM index (step (2) in Fig. 1): First, the flash index is read as preliminary RAM index. Then all changes that were recorded in the log are applied to this preliminary RAM index. After the replay, the correct RAM index has been rebuilt.

3 Data Layout for UBIFS

The data layout used in UBIFS follows the file system representation used in the Linux virtual file system switch (VFS). The Linux kernel implements POSIX file system functions [15] as calls to VFS functions that provide a common interface for all implemented file systems.

Inodes (index nodes) are the primary data structure used in VFS[2]. They represent objects in the file system (such as files, directories, symlinks or devices) and are identified by an inode number. Inodes store information about objects, such as size, link count, modification date or permissions, but do not contain the name of the object. Mapping between names and objects is done using *dentries* (directory entries) which say that a certain object can be found in a certain directory under a certain name[3]. This separation is required as a single inode can be referred to in multiple directories (used for hard links). The directory tree can be viewed as an edge-labeled, directed graph consisting of inodes as vertices and dentries as edges. Furthermore negative dentries are used in memory to express that no object with the given name exists in a directory. These are used as a response when looking for nonexistent files, or as parameters for the file name when creating new files or directories. File contents are stored as fixed-size data blocks, called *pages*, belonging to file inodes. When opening a file or directory to access its contents, a *file* data structure (called file handle in user space) is used in memory to manage the inode number and the current position within the inode.

---

[1] This can be performed atomically, because all changes are stored out-of-place and the effective replace is executed by updating pointers to the data structures.

[2] See `struct inode` in `include/linux/fs.h`, [19]

[3] See `struct dentry` in `include/linux/dcache.h`, [19]

We thus define inodes, dentries and file handles as free data types generated by constructors mkinode, mkdentry, negdentry and mkfile.

inode = mkinode (. .ino : nat; . .directory : bool; . .nlink : nat; . .size : nat)
dentry = mkdentry (. .name : string; . .ino : nat) **with** . .dentry?
    | negdentry (. .name : string) **with** . .negdentry?
file = mkfile (. .ino : nat; . .pos : nat)

This definition includes selector functions `.ino`, `.directory`, `.nlink`, `.size`, `.name` and `.pos` for accessing the constructor arguments, as well as type predicates `.dentry?` and `.negdentry?` to decide between the two types of dentries (the dot before predicates and functions indicates postfix notation).

UBIFS stores these data structures (except for file handles and negative dentries which are never stored) as nodes as described in the previous section. These nodes contain meta data (called *key*) to uniquely identifiy the corresponding node, and data containing the remaining information. For inodes, the inode number is sufficient as a key, whereas dentries require the parent inode number and the file name. Pages are referred to by the inode number and the position within the file.

Figure 2 shows the representation of a sample directory tree as UBIFS nodes. It contains two files, `test.txt` and `empty.txt` and a directory `temp` containing a hard link to `test.txt` named `test2.txt`.

Directory Tree

Name	*Inode*
[ROOT]	1
├ test.txt	2
├ empty.txt	4
└ temp/	3
└ test2.txt	2

UBIFS Representation (Nodes)

INODENODE	INODEKEY(1)	directory: true, nlink: 3, size: 2
DENTRYNODE	DENTRYKEY(1, test.txt)	name: „test.txt“, ino: 2
DENTRYNODE	DENTRYKEY(1, empty.txt)	name: „empty.txt“, ino: 4
DENTRYNODE	DENTRYKEY(1, temp)	name: „temp“, ino: 3
INODENODE	INODEKEY(2)	directory: false, nlink: 2, size: 2
DATANODE	DATAKEY(2, 1)	PAYLOAD DATA
DATANODE	DATAKEY(2, 2)	PAYLOAD DATA
INODENODE	INODEKEY(3)	directory: true, nlink: 2, size: 1
DENTRYNODE	DENTRYKEY(3, test2.txt)	name: „test2.txt“, inode: 2
INODENODE	INODEKEY(4)	directory: false, nlink: 1, size: 0

Fig. 2. Directory tree representation in UBIFS

Nodes for inodes in this abstraction contain extra information about size and link count. For files, the size gives the file size, measured in number of pages, and links gives the number of dentries (hard links) referencing the inode. Directories use the number of contained objects as size, and the number of links is calculated as (2 + number of subdirectories)[4].

[4] Directories may not appear as hard links, so this number is the result of counting the one allowed link, the "virtual" hard link "`.`" of the directory to itself and the "`..`" link to the parent in each subdirectory.

For nodes and keys, we use the following specification[5]:

node = inodenode (. .key : key; . .directory : bool;
. .nlink : nat; . .size : nat) **with** . .inode?
| dentrynode (. .key : key; . .name : string;
. .ino : nat) **with** . .dentry?
| datanode (. .key : key; . .data : page) **with** . .data?
key = inodekey (. .ino : nat) **with** . .inode?
| datakey (. .ino : nat; . .part : nat) **with** . .data?
| dentrykey (. .ino : nat; . .name : string) **with** . .dentry?

Information about inode 1 can be found in a node with inode key 1. To list all objects contained in this directory, all (valid) nodes with a dentry key containing 1 as a first argument have to be enumerated. The same goes for reading contents of a file, by scanning for corresponding data keys (their first parameter denotes the inode number, the second states the position inside the file).

File system data cannot directly be indexed and accessed using flash memory locations, as this would require overwriting flash memory on data change. Instead, data is referred to by its unique key. Finding matching nodes for a given key by sequentially scanning the entire flash memory however is very slow. UBIFS holds an index datastructure mapping keys to flash memory addresses in the RAM (called *RAM index*). It allows to quickly access nodes with a given key, or to enumerate all existing dentry or data keys for a given inode. When writing a new version of an existing node, the new copy is written to the flash store, and the address for its key is updated in the RAM index.

Formally, both the flash store and the two indexes (in RAM and on flash) are an instance of the abstract data type *store(elem,data)*.

flashstore = store(address,node) **index** = store(key,address)

A store is a partial function with a finite domain of elements with type *elem* and codomain *data*. In a set-theory based language such as Z [27] stores would be represented as a relation (set of pairs) for the function graph, and the update operation would be written as $st \oplus \{k \mapsto d\}$. In KIV stores are directly specified as an abstract, non-free data type generated from the empty store and an update operation *st[k,d]*, which allocates k if necessary, and overwrites the value at k with d. This avoids the need for a left-uniqueness constraint as an invariant.

4 The UBIFS journal

This section describes how the journal operates and how it is linked to flash and RAM index. The correlation between flash store, journal and the indices is shown in Fig. 3.

[5] See `struct ubifs_data_node`, `struct ubifs_ino_node` and `struct ubifs_data_node` in `fs/ubifs/ubifs-media.h` for nodes, and `struct ubifs_key` in `fs/ubifs/ubifs.h` as well as `ino_key_init`, `dent_key_init` and `data_key_init` in `fs/ubifs/key.h`, [19]

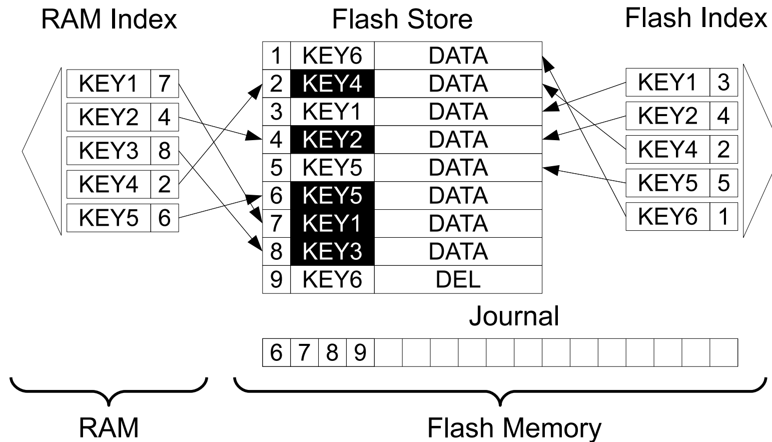

Fig. 3. RAM index, flash index, flash store and journal

To create a new node, this is written to an unused address in the flash store and simultaneously[6] added to the log. Afterwards, its address is stored in the RAM index for further access to its key. This way, the data is safe, even if the RAM index gets lost without a commit e. g. caused by a power failure, because the correct state of the RAM index can be restored by information from flash index, flash store and log.

This method allows for creating new and overwriting existing nodes. However, deleting nodes is not possible because it would require pure index operations (delete a key from the index). Therefore, UBIFS uses specialized delete nodes which are written to the flash store, but cause deletion from the RAM index when replayed (marked as DEL[7] in Fig. 3).

When performing a replay in the situation of Fig. 3, the contents of the flash index are copied to the RAM index. When replaying address `6`, the `6` is stored in the RAM index as new address for key `KEY5`. The same goes for address `7`, while `8` adds `KEY3` to the index. Address `9` contains a deletion entry that causes `KEY6` to be deleted from the index.

The figure also shows the need for garbage collection: addresses `1`, `3` and `5` store data which are no longer in the index and therefore can be reused.

5 File System Operations

For applying changes to the contents of the file system, the Linux VFS provides file system operations. These can be grouped into inode operations, file opera-

[6] This is possible because UBIFS does not use an explicit list for the log, but treats all nodes in certain LEBs as log entries.

[7] UBIFS uses inode nodes with link count 0 or dentry nodes with destination inode number 0 to delete the corresponding keys.

tions, and address space operations. Inode operations allow creating, renaming or deleting inodes, whereas file operations allow for working with inode contents (data stored in files or directory entries). Address space operations include all operations that work with pages, and are used in the Linux kernel to implement file operations accessing file contents. They are included here to allow for using abstract pages when handling file contents.

We will start with an inode operation called *create*[8], used for creating new files. It expects the inode number of the containing directory (`P_INO`), and a negative dentry specifying the file name (`DENT`) as input. `DENT` is overwritten with the newly created dentry.

```
create#(P_INO; DENT, FS, RI, LOG) {
  choose INO with ¬ inodekey(INO) ∈ RI ∧ INO > 0 in {
     let INODE = getinode(P_INO, FS, RI) in
     choose ADR1, ADR2, ADR3 with new(ADR1, ADR2, ADR3, FS) in {
        FS := FS
          [ADR1, inodenode(inodekey(INO), false, 1, 0)]
          [ADR2, dentrynode(dentrykey(P_INO, DENT.name), DENT.name, INO)]
          [ADR3, inodenode(inodekey(INODE.ino),
             INODE.directory, INODE.nlink, INODE.size + 1)],
        LOG := LOG + ADR1 + ADR2 + ADR3;
        RI := RI[inodekey(INO), ADR1];
        RI := RI[dentrykey(P_INO, DENT.name), ADR2];
        RI := RI[inodekey(INODE.ino), ADR3]  };
     DENT := mkdentry(DENT.name, INO)  }};
```

The notation used to describe the rule is similar to that of ASM rules [11], [3], but it should be noted that only parallel assignment, denoted with a comma, is executed atomically, while sequential composition (with semicolon) is not. **choose** binds new local variables (here e.g. INO) to values that satisfy the **with** clause.

The operation writes a new inode node for the created file (link count `1`, size `0`) and a dentry node for a dentry pointing from the parent directory `P_INO` to the new inode, named as given in `DENT`. It increases the parent directory size by 1 to reflect the increased number of objects contained in the directory.

To correctly perform these changes, it first selects an unused inode number and three new addresses from the flash store, and loads the inode given by `P_INO`. It then atomically writes three new nodes (predicate new) into new locations of the flash store `FS`, simultaneously adding the locations to the journal `LOG`. Afterwards it updates the RAM index `RI` with the new addresses, and changes the reference parameter `DENT` to return the newly created dentry.

The following inode operations also change the directory structure. Their informal description leaves out standard parameters FS, RI and LOG. Full details can be found on the Web [18].

unlink(P_INO, DENT) Removes the file referred to by `DENT` from the directory `P_INO`. If the dentry was the last link to the referred file, the inode and file

[8] See `ubifs_create` in `fs/ubifs/dir.c`, [19]

contents are also deleted, otherwise only the dentry is removed. `DENT` is returned as a negative dentry.

link(OLD_DENT, NEW_INO, NEW_DENT) Creates a hard link to the file referred to by `OLD_DENT`, placed in the directory `NEW_INO` and named as given by the negative dentry `NEW_DENT`. Returns the newly created dentry in `NEW_DENT`.

mkdir(P_INO, DENT) Creates a new directory in `P_INO`, with the name given in the negative dentry `DENT`. The newly created dentry is returned in `DENT`.

rmdir(P_INO, DENT) Removes the (empty) directory referred to by the dentry `DENT` located in the parent directory `P_INO`. `DENT` is changed into a negative dentry.

rename(OLD_INO, OLD_DENT, NEW_INO, NEW_DENT) Moves the object (file or directory) referred to by `OLD_DENT` from directory `OLD_INO` to directory `NEW_INO`, changing its name to `NEW_DENT.name`. If the object referred to by `NEW_DENT` exists, it has to be of the same type (file or directory) as `OLD_DENT`, and it is overwritten (i. e. deleted).

lookup(P_INO, DENT) Checks for the existence of a object named `DENT.name` in the directory `P_INO`. If it exists, the dentry is returned in `DENT`, otherwise a negative dentry is returned.

For inode contents, the following file and address space operations exist:

open(INO, FILE) Opens the file or directory given in `INO`, and returns a new file handle in `FILE`.

release(INO, FILE) Called when the last process closes an inode (file or directory), to clean up temporary data. Unused in the given specification.

readpage(FILE, PAGENO, PAGE) Reads the page with number `PAGENO` from the file referred to in `FILE`, and returns it in `PAGE`.

writepage(FILE, PAGENO, PAGE) Writes the data from `PAGE` as new page numbered `PAGENO` to file `FILE`.

truncate(FILE, PAGENO) Sets the file size of the file referred to in `FILE` to `PAGENO`, deleting all pages beyond.

readdir(FILE, DENT) Returns the next object of the directory referred to in `FILE`, or a negative dentry if no further file or directory exists. The (positive or negative) dentry is returned in `DENT`, and the position stored in `FILE` is increased to return the next object at the next call.

Finally, our model defines garbage collection, commit and replay as described in Sect. 2.

6 Verification

Our verification efforts have focussed on three properties, described in the following paragraphs. The last paragraph gives a summary of the effort involved.

Functional Correctness of the Operations. We proved that all specified operations terminate and fulfill postconditions about their results. As most operations and all supporting functions are non-recursive and only use loops over the elements of finite lists, termination is quite obvious, and proving does not pose great complexity – even regardless whether any preconditions hold or not.

Only garbage collection has a precondition for termination, as it is specified as a non-deterministic choice of a new, isomorphic state, which only terminates if such a state exists.

For the other inode and file operations, we give and prove total correctness assertions that describe their behaviour. We write $wp(\alpha,\varphi)$ to denote the weakest precondition of program α with respect to a postcondition φ[9]. Proofs in KIV are done using sequent calculus and symbolic execution of programs, see [23] for details.

For the create operation described in the previous section we demand[10].

$$\begin{array}{l}\text{valid-dir-ino(P)} \wedge \text{valid-negdentry(P, DENT)} \\ \rightarrow \mathit{wp}(\text{create(P; DENT), valid-dentry(P, DENT)} \wedge \text{valid-file-ino(DENT.ino))}\end{array}$$

When called with a valid directory inode (i. e. the inode exists, is of type directory and has a link count of 2 or higher) and a valid negative dentry (i. e. no dentry with the given name exists in the given directory) as parameters, the operations yield a dentry (ideally with the same name, but we do not demand that here) that exists in the given directory and references a valid file (i. e. the inode referred to exists, is a file and has a link count of at least 1).

Giving postconditions for readpage or writepage individually turned out to be rather hard when trying to remain implementation independent, so we decided to use the combined postcondition that reading data after writing returns exactly the data written:

$$\mathit{wp}(\text{writepage(file, pageno, pg) ; readpage(file, pageno; pg2), pg} = \text{pg2})$$

Furthermore, file contents of a file remain unchanged when writing to another file or to another position inside the file:

$$\begin{array}{l}\text{valid-file(f1)} \wedge \text{valid-file(f2)} \wedge (\text{f1.ino} \neq \text{f2.ino} \vee \text{n1} \neq \text{n2}) \wedge \text{store-cons(fs,ri,fi,log)} \\ \rightarrow \mathit{wp}(\text{readpage\#(f1,n1;p1);writepage\#(f2,n2,p);readpage\#(f1,n1;p2), p1=p2})\end{array}$$

The pre- and postconditions for the other operations as well as their proofs can be found on the Web [18].

Consistency of the File System. Another basic requirement is that the file system is always consistent. We formally define a predicate *fs-cons(fs,ri)* for the file store *fs* and the RAM index *ri* (flash index and log are irrelevant as they are only required for replay), and prove that it is an invariant of all operations.

For each key stored in the RAM index, *fs-cons* requires that its address must be allocated in the flash store, and that the key is stored as that node's key. Further requirements depend on the type of key.

Dentry keys must belong to a valid directory and reference a valid inode. The name stored in the key must be equal to the copy of the name stored in the node. Data keys have to belong to a valid file inode, and the requirements for inode keys are differentiated between files and directories. For files, the link count has to be equal to the number of dentries referencing the file, and for each

[9] In KIV $wp(\alpha,\varphi)$ is written as $\langle\!|\alpha|\!\rangle\varphi$.

[10] We suppress standard parameters FS, RI and LOG in all predicates and procedure calls for better readability.

data key belonging to the file, the page number (*part*) has to be less than the file's size. Directories have to have 2 + number of subdirectories as their link count, and the number of contained dentries as size. Furthermore, no directory may have more than 1 (stored) dentry referencing it[11].

The formal proof obligation for invariance is

$$\text{fs-cons(fs, ri)} \rightarrow wp(op, \text{fs-cons(fs, ri)})$$

where *op* stands for any of the operations defined in the previous section. As this property describes correlations between the different types of keys, it cannot be proven step by step for each individual update of flash store and RAM index; the property is not restored until all steps of an operation are completed. So the proofs have to take the operation as a whole, complicating the definition and application of reusable lemmata.

Correctness of the Replay Process. The replay operation should be able to restore a consistent state after a crash, losing as little data as possible. We therefore define a predicate *log-cons* claiming that a replay in the current situation will correctly restore the RAM index to a state isomorphic to the one with current RAM index contents. The formal definition is

$$\text{log-cons(fs, ri, fi, log)} \leftrightarrow wp(\text{replay(fs, fi, log, ri2), (fs, ri)} \cong \text{(fs, ri2)})$$

If this predicate is true, we will not lose data at a crash (except maybe for the changes of the current operation). A reliable file system should always preserve this predicate, even in the middle of an operation. For verification we have taken the weaker approach to prove that this predicate is invariant

$$\begin{aligned} &\text{log-cons(fs,ri,fi,log)} \wedge \text{store-cons(fs,ri,log)} \wedge \text{datanode-cons(fs,ri)} \\ \rightarrow\ &wp(op, \text{log-cons(fs,ri,fi,log)} \wedge \text{store-cons(fs,ri,log)} \wedge \text{datanode-cons(fs,ri)}) \end{aligned}$$

Note that *log-cons* used in the pre- and postcondition is defined using a wp-formula itself, so the formula is not a total correctness formula in the usual sense, where pre- and postcondition are defined using predicate logic only. Nevertheless KIV's logic can prove this formula using symbolic execution for formulas in preconditions too.

The invariance above ensures the file system robust wrt. crashes *between* operations. Still, the implementation of the operations is designed in a way such that a similar property also holds anytime during the execution of operations. As one of the next steps we plan to prove this fact using KIV's temporal logic [1], [2] which is able to express and prove the full property[12].

To prove the invariance of *log-cons* required two auxiliary invariants, *store-cons* and *datanode-cons*. The predicate store-cons requires that each address referred to in the RAM index or log has to be allocated in the flash store, and datanode-cons demands that each data key belongs to a valid file inode and

[11] The root directory has no link, all other directories have one, as further hard links to directories are not allowed.

[12] An alternative would be to encode the operations as a sequence of small steps using a program counter, as is often done for model checking. Then the property would have to be proved to be invariant in every small step.

describes a page within the file length. The former is needed to avoid accessing addresses in the flash store that are not yet allocated, whereas the latter is needed as replaying some operations causes data keys beyond the file size to be deleted.

Statistics about our Specification and Verification. Developing and verifying our specification mainly consisted of four steps. We first collected and analyzed material about UBIFS, mainly from the UBIFS whitepaper [14] and the UBIFS source code in the Linux kernel. During four weeks, we developed a basic understanding of the mechanisms and found a suitable abstraction level for the specification. In the following two weeks, the required data structures and operations were specified, as well as the invariants we wanted to prove. Proving the correctness of the replay operation (log-cons) took another two weeks, during which we corrected minor errors in the specification and found the additional preconditions needed for log consistency. Our last steps were to prove the total correctness assertions and the file system consistency. This took about as long as the log consistency, though the resulting proofs for fs-cons were a lot larger than the ones for log-cons – especially for the *rename* operation which contains many case distinctions (file vs. directory, rename vs. overwrite).

7 Outlook

The work of this paper defines a first abstract model of the essential data structures needed in a FFS. We intend to use it as an intermediate layer in the development of a sequence of refinements, which starts with an abstract POSIX specification such as the ones of [5], [7] and ends with an implementation based on a specification of flash memory, like the one of [4]. There will be many obstacles deriving such refinements, and we discuss these problems briefly in the following paragraphs on future work.

A rather different approach has been taken by Kang and Jackson [17]. This work builds a vertical prototype by focussing on the read/write operations for files, ignoring issues such as directory structure, indexes and journals (their file system roughly corresponds to the file store component of our model). They define an abstract level, where reading/writing a file is atomic. These are refined to reading and writing pages, which is done on a model that is closer to implementation than ours, since it already considers a mapping from logical to physical blocks. The model also includes an interesting analysis of a high-level recovery mechanism for failed write attempts. UBIFS delegates recovery mechanism mainly to UBI (see below). Two high-level mechanisms for recovery exist in UBIFS: one for log entries, which may not have been completely written; another for the root node of the B^+-tree of the file index, see [14]. Both are very different from the one analyzed in [17]. Since our model is still too abstract (no B^+-trees and no concrete layout of the journal as data in erase blocks), these will only show up in refinements.

To check properties of the model, Alloy (and the Kodkod engine) is used to check that finite approximations of the model do not have counter examples.

This approach is weaker than verification, but gives excellent results for debugging specifications. Our current work is on attaching Kodkod as a pre-checker to validate KIV theorems before proof attempts [6], similar to the proposal in [7].

From POSIX to our UBIFS Model. Our abstract model is based on the interface that UBIFS offers to the general Linux virtual file system switch (VFS). It assumes that our operations are protected by locks and will therefore not be executed concurrently. This is mostly true in the implementation, with one notable exception: the commit operation may be executed in parallel with regular operations. However, above our specification this is no longer true. As an example, writing a full file can no longer be assumed to be atomic: it will consist of several pages being written (it is possible that several processes concurrently read and write a file!). In reality, these page writes will even be cached. Even if a write operation has finished, the data may not yet be on the flash (the Linux flush command ensures that caches are emptied). We therefore expect the theoretical question of how concurrency should be handled to dominate the question of a correct refinement.

As a first step, it is of course possible to ignore the concurrency problem (as has been done in [17]). Then implementing POSIX operations correctly using our abstract interface should be possible using standard data refinement. Of course, for such a refinement, some additional data such as modification dates and access rights would have to be added.

From our Model to Flash Memory. Our model has abstracted from many details of a real flash file system. First, and most important we have abstracted from wear leveling. Since wear leveling is not dealt within UBIFS, but in a separate UBI layer that maps logical to physical erase blocks, this seemed natural. We expect the correctness of this layer not to pose too difficult theoretical questions. The challenging question for this refinement is whether it is possible to prove something about the quality of wear leveling. This looks possible for UBI, since its wear leveling strategy is based on counting erase cycles.

Second, we have abstracted index structures, which are B^+-trees in reality. The lower level representation allows two optimizations: first, only those parts of the flash index which are currently needed, must be loaded into RAM. Second, the commit operation does not simply copy the full B^+-tree from RAM to the flash memory as in our simple specification. Instead it copies only those parts that have changed since the last commit. This means that the flash index becomes a "wandering tree". Parts of it move with every commit.

Third, all three data structures, the flash store, the flash index and the journal will be represented uniformly by pieces of memory in logical erase blocks (LEBs). The challenging problem here is to verify garbage collection, which we only specified to give some isomorphic file system. This algorithm is rather complex. It uses an auxiliary data structure, to find out efficiently how much room is left in each LEB. This data structure, the LPT ("LEB property tree") is also implemented as a wandering tree.

Finally, there are several more issues which we have ignored: on the fly compression (using zlib and LZO) and the handling of orphan nodes, which are

needed to handle still open files that have been deleted, or hashing of index keys are three examples.

In summary, we think that the development of a verified flash file system will need a lot more effort than our previous contribution to the Grand Challenge with the Mondex case study ([12], [25], [24], [10]).

8 Conclusion

We have given an abstract specification of a flash file system which was derived by abstracting as much as possible from the details of the UBIFS system. We have specified the four central data structures: the file store which stores node-structured data, the flash index, its cached copy in the RAM and the journal. Based on these, we have specified the most relevant interface operations.

We have verified that the operations keep the file system in a consistent state, and that they satisfy some total correctness assertions. We have also verified that the journal is used correctly and enables recovery at every time.

Our model should be of general interest for the development of a correct flash file system, since variants of the data structures and operations we describe should be relevant for every realistic, efficient implementation.

Nevertheless the model given in this paper is only our first step towards the development of a verified flash file system implementation. We plan to use the model as an intermediate layer of a series of refinements, which starts with an abstract model of POSIX-like operations and leads down to to an efficient implementation like UBIFS based on a specification of flash memory.

References

1. M. Balser, S. Bäumler, W. Reif, and G. Schellhorn. Interactive verification of concurrent systems using symbolic execution. In *Proceedings of 7th International Workshop of Implementation of Logics (IWIL 08)*, 2008.
2. S. Bäumler, F. Nafz, M. Balser, and W. Reif. Compositional proofs with symbolic execution. In B. Beckert and G. Klein, editors, *Proceedings of the 5th International Verification Workshop*, volume 372 of *Ceur Workshop Proceedings*, 2008.
3. E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
4. A. Butterfield and J. Woodcock. Formalising flash memory: First steps. In *Proc. of the 12th IEEE Int. Conf. on Engineering Complex Computer Systems (ICECCS)*, pages 251–260, Washington DC, USA, 2007. IEEE Comp. Soc.
5. K. Damchoom, M. Butler, and J.-R. Abrial. Modelling and proof of a tree-structured file system in Event-B and Rodin. In *Proc. of the 10th Int. Conf. on Formal Methods and Sw. Eng. (ICFEM)*, pages 25–44. Springer LNCS 5256, 2008.
6. A. Dunets, G. Schellhorn, and W. Reif. Automating Algebraic Specifications of Non-freely Generated Data Types. In *ATVA*, LNCS 5311, pages 141 –155, 2008.
7. M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying Intel flash file system core specification. In *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*. School of Computing Science, Newcastle University, 2008. Technical Report CS-TR-1099.

8. L. Freitas, J. Woodcock, and A. Butterfield. Posix and the verification grand challenge: A roadmap. In *ICECCS '08: Proc. of the 13th IEEE Int. Conf. on Eng. of Complex Computer Systems*, pages 153–162, Washington, DC, USA, 2008.
9. E. Gal and S. Toledo. Algorithms and data structures for flash memory. *ACM computing surveys*, pages 138–163, 2005.
10. H. Grandy, M. Bischof, G. Schellhorn, W. Reif, and K. Stenzel. Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code. In *FM 2008: 15th Int. Symposium on Formal Methods*. Springer LNCS 5014, 2008.
11. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford Univ. Press, 1995.
12. D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. *Formal Aspects of Computing*, 20(1), January 2008.
13. C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
14. A. Hunter. A brief introduction to the design of UBIFS. URL: `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`, 2008.
15. The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, 2004. URL: `http://www.unix.org/version3/online.html` (login required).
16. R. Joshi and G. J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2), June 2007.
17. E. Kang and D. Jackson. Formal modelling and analysis of a flash filesystem in Alloy. In *Proceedings of ABZ 2008*, pages 294 – 308. Springer LNCS 5238, 2008.
18. Web presentation of the Flash File System Case Study in KIV, 2009. URL: `http://www.informatik.uni-augsburg.de/swt/projects/flash.html`.
19. LXR - the Linux cross reference. URL: `http://lxr.linux.no/`.
20. C. Morgan and B. Sufrin. Specification of the UNIX filing system. In *Specification case studies*, pages 91–140. Prentice Hall (UK) Ltd., Hertfordshire, UK, 1987.
21. J.N. Oliveira. Extended Static Checking by Calculation Using the Pointfree Transform. In *LerNet ALFA Summer School 2008*, Springer LNCS 5520, 2008.
22. G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference, 2005 IEEE*, pages 4186–4199, March 2005.
23. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II, chapter 1, pages 13 – 39. Kluwer Academic Publishers, Dordrecht, 1998.
24. G. Schellhorn and R. Banach. A concept-driven construction of the mondex protocol using three refinements. In *Proc. of ABZ 2008*. LNCS 5238, 2008.
25. G. Schellhorn, H. Grandy, D. Haneberg, N. Moebius, and W. Reif. A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In U. Glässer J.-R. Abrial, editor, *Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*. Springer LNCS 5115, 2008.
26. G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll. Verified Formal Security Models for Multiapplicative Smart Cards. *Special issue of the Journal of Computer Security*, 10(4):339 – 367, 2002.
27. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.
28. STMicroelectronics. *SPC56xB/C/D Automotive 32-bit Flash microcontrollers for car body applications*, February 2009. URL: `http://www.st.com/stonline/products/promlit/a_automotive.htm`.
29. D. Woodhouse. JFFS : The Journalling Flash File System. URL: `http://sources.redhat.com/jffs2/jffs2.pdf`, 2001.