Development of a Verified Flash File System [⋆]

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

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

Abstract. This paper gives an overview over the development of a formally verified file system for flash memory. We describe our approach that is based on Abstract State Machines and incremental modular refinement. Some of the important intermediate levels and the features they introduce are given. We report on the verification challenges addressed so far, and point to open problems and future work. We furthermore draw preliminary conclusions on the methodology and the required tool support.

1 Introduction

Flaws in the design and implementation of file systems already lead to serious problems in mission-critical systems. A prominent example is the Mars Exploration Rover Spirit [34] that got stuck in a reset cycle. In 2013, the Mars Rover Curiosity also had a bug in its file system implementation, that triggered an automatic switch to safe mode. The first incident prompted a proposal to formally verify a file system for flash memory [24,18] as a pilot project for Hoare's Grand Challenge [22].

We are developing a verified flash file system (FFS). This paper reports on our progress and discusses some of the aspects of the project. We describe parts of the design, the formal models, and proofs, pointing out challenges and solutions.

The main characteristic of flash memory that guides the design is that data cannot be overwritten in place, instead space can only be reused by erasing whole blocks. Therefore, data is always written to new locations, and sophisticated strategies are employed in order to make the file system efficient, including indexing and garbage collection. For the algorithmic problems, we base our design on UBIFS [23,20], which is a state-of-the-art FFS in the Linux kernel.

In order to tackle the complexity of the verification of an entire FFS, we refine an abstract specification of the POSIX file system interface [41] in several steps down to an implementation. Since our goal is an implementation that runs on actual hardware, besides functional correctness additional, nontrivial issues must be addressed, such as power cuts and errors of the hardware. A further requirement is that the models must admit generation of executable code. The

[⋆] The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-662-43652-3_2 derived code should correspond very closely to the models, thus simplifying an automated last refinement proof between the models and the actual code.

The paper is structured as follows: Section 2 gives an overview over our approach and models. Section 3 provides more detail on the design of each of the models and their concepts. The section also highlights some of the problems that need to be tackled and describes our solutions. The formal models are also available at our web-presentation [11]. Section 4 explains how we generate Scala and C code from our model hierarchy and integrate it into Linux. In Sec. 5 we report on some of the lessons learned during the project. Section 6 concludes and points out some of the open challenges.

2 Overview

Figure 1 shows the high-level structure of the project. There are four primary conceptual layers. A top-level specification of the POSIX file system interface [41] defines functional correctness requirements. At the bottom is a driver interface model that encodes our assumptions about the hardware. Two layers in between constitute the main functional parts of the system: The file system (FS) implementation is responsible for mapping the high-level concepts found in POSIX (e.g., directories, files and paths) down to an on-disk representation. A separate layer, the erase block management (EBM), provides advanced features on top of the hardware interface (e.g., logical blocks and wear-leveling).

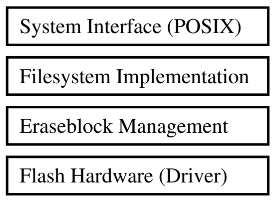

Fig. 1. High-level structure

Figures 2 and 3 show how the file system implementation is broken down to a refinement hierarchy. Boxes represent formal models. Refinements—denoted by dashed lines—ensure **functional correctness** of the final model that is the result of combining all models shaded in grey. From this combined model we generate executable Scala and C code.

The POSIX model for example is refined by a combination of a Virtual Filesystem Switch (VFS) and an abstract file system specification (AFS). The former realizes concepts common to all FS implementations such as path traversal. Such a VFS exists in Linux and other operating systems ("Installable File System" in Windows). The benefit of this approach is that concrete file systems such as UBIFS, Ext or FAT do not have to reimplement the generic functionality. Instead they satisfy a well-defined internal interface, denoted by the symbol –◎– in Fig. 2. We capture the expected functional behaviour of a concrete FS by the AFS model. The modularization into VFS and AFS significantly reduces the complexity of the verification, since the refinement proof of POSIX to VFS is independent of the actual implementation, whereas the implementation needs to be verified against AFS only. The top-level layers are described in more detail in Sections 3.1 and 3.2.Although we provide our own verified VFS implementation, it will be possible to use the file system core with the Linux VFS instead.

Flash memory is not entirely reliable, for example, read operations may fail nondeterministically. Hence, **error handling** is taken into account from the beginning. Concretely, the formal POSIX specification admits such errors as long as an unsuccessful operation has no observable effect. However, on intermediate layers failures may be visible.

A major concern that is orthogonal to functional correctness is **reset safety**: the file system guarantees recovery to a well-defined state after an unexpected power-cut. Reset-safety must be considered at different levels of abstraction. It is also critical that power-cuts are considered *during* an operation.

For example, the LogFS model introduces a distinction between persistent data stored on flash and volatile data in RAM. Only the latter is lost during power-cuts. New entries are first stored in a log and only later committed to flash. We prove abstractly that after a crash the previous RAM state can be reconstructed from the entries in the log and the flash state, as described in detail in Sec. 3.3. Atomicity of writes of several nodes is provided by another layer (transactional journal).

The core of the flash file system (represented by the correspondingly named box) implements strategies similar to UBIFS. This layer was also the first we have looked at [39]. The idea of UBIFS is to store a collection of data items, called "nodes", that represent files, directories, directory entries, and pages that store the content of files. This collection is unstructured and new data items are always written to a fresh location on flash, since overwriting in-place is not possible. Current versions of a node are referenced by an efficient **index**, implemented as a B^+ tree. The index exists both on flash and in memory. The purpose of the flash index is to speed up initialization of the file system, i.e., it is loaded on-demand. It is not necessary to scan the whole device during boot-up. Over time unreferenced nodes accrue and **garbage collection** is performed in the background to free up space by erasing the corresponding blocks.

Updates to the flash index are expensive due to the limitations of the hardware. Hence, these are collected up to a certain threshold and flushed periodically by a commit operation; one accepts that the flash index is usually outdated.

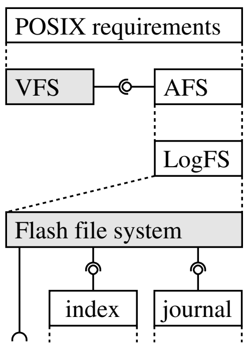

Fig. 2. Upper Layers of the File System

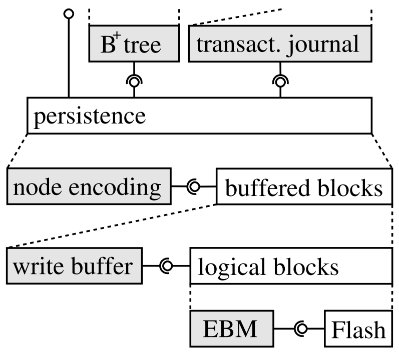

Fig. 3. Lower Layers of the File System

Therefore, it is of great importance, that all information kept in RAM is actually redundant and can be recovered from flash. For this purpose, new data is kept in a special area, the *log*. The recovery process after a crash then reads the log to restore the previous state. In Sec. 3.3 we show an abstract version of this property. The log is part of the journal module in Fig. 2.

The nodes of the FFS have to be stored on the flash device. This is accomplished by the persistence layer. For space and time efficiency, the writes are buffered until an entire flash page can be written. Sec. 3.4 describes the problems that arise when writes are cached.

Besides correctness and reset safety, it is important that the physical medium is used evenly, i.e., erase cycles are distributed approximately equally between the blocks of the flash device, because erasing physically degrades the memory. To prolong the lifetime of the device, a technique called **wear-leveling** is implemented by the erase block management. Section 3.5 describes this layer.

3 Models

In this section we outline several formal models. There is a general schema. *Abstract* models, such as POSIX and AFS are as simple as possible. *Concrete* models, namely all grey components, are rather complex, in contrast.

Our tool is the interactive theorem prover KIV [35,12]. For a detailed description of the specification language see for example [13] (this volume) and [38]. It is based on *Abstract State Machines* [5] (ASMs) and ASM refinement [4]. We use algebraic specifications to axiomatize data types, and a weakest-precondition calculus to verify properties. This permits us to use a variety of data types, depending on the requirements. In the case study, lists, sets, multisets and maps (partial functions $\tau \nrightarrow \sigma$, application uses square brackets $f[a]$) are typically used in more abstract layers, whereas arrays are prevalent in the lower layers.

3.1 POSIX Specification and Model

Our formal POSIX model [15] defines the overall requirements of the system. It has been derived from the official specification [41] and is furthermore an evolution of a lot of existing related work to which we compare at the end of this section. Our model has been developed with two goals in mind:

1. It should be as abstract as possible, in particular, we chose an algebraic tree data structure to represent the directory hierarchy of the file system. The contents of files are stored as sequences of bytes.
2. It should be reasonably complete and avoid conceptual simplifications in the interface. Specifically, we support hard-links to files, handles to open files, and orphaned files (which are a consequence of open file handles as explained below); and the model admits nondeterministic failures in a well-specified way. These concepts are hard (or even impossible) to integrate in retrospect.

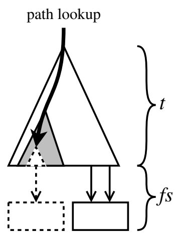

Fig. 4. FS as a tree

```
posix_create($path, md; err$)
  choose $err'$
  with pre-create($path, md, t, fs, err'$)
  in $err$ := $err'$

  if $err$ = ESUCCESS then
    choose $fid$ with $fid \notin fs$
    in $t[path]$   := fnode($fid$)
       $fs[fid]$   := fdata($md, \langle\rangle$)
```

Fig. 5. POSIX create operation

The model is easy to understand but also ensures that we will end up with a realistic and usable file system. Formally, the state of the POSIX model is given by the directory tree $t : Tree$ and a file store $fs : Fid \nrightarrow FData$ mapping file identifiers to the content.

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

Structural system-level operations are defined at the level of *paths*. As an example, Fig. 4 shows the effect of `create`, where the grey part denotes the parent directory, and the parts with a dotted outline are newly created. The corresponding ASM rule is listed in Fig. 5.

The operation takes an absolute path *path* and some metadata *md* as input. After some error handling (which we explain below), a fresh file identifier *fid* is selected and stored in a file node (`fnode`) at path *path* in the tree. The file store *fs* is extended by a mapping from *fid* to the metadata and an empty content, where $\langle\rangle$ denotes an empty sequence of bytes. File identifiers serve as an indirection to enable hard-links, i.e., multiple references to the same file from different parent directories under possibly different names. An error code *err* is returned.

The converse operation is `unlink`($path; err$), which removes the entry specified by *path* from the parent directory. In principle, it also deletes the file's content as soon as the last link is gone. File content can be read and written sequentially through *file handles fh*, which are obtained by `open`($path; fh, err$) and released by `close`($fh; err$). As a consequence, files are not only referenced from the tree, but also from open file handles, which store the identifier of the respective file. The POSIX standard permits to unlink files even if there is still an open file handle pointing to that file. The file's content is kept around until the last reference, either from the directory tree or a handle, is dropped.

Files that are no longer accessible from the tree are called *orphans*. The possibility to have orphans complicates the model and the implementation (see Sec. 3.3 on recovery after power-failure). However, virtually all Linux file systems provide this feature, since it is used by the operating system during package updates (application binaries are overwritten in place, while running applications still reference the previous binary image through a file handle) and by applications (e.g., Apache lock-files, MySQL temporary files).

Each POSIX operation returns an error code that indicates whether the operation was successful or not. Error codes fall into two categories. Preconditions are rigidly checked by the POSIX model in order to protect against unintended or malicious calls. For each possible violation it is specified which error codes may be indicated, but the order in which such checks occur is not predetermined. The other type of errors may occur anytime and corresponds to out-of-memory conditions or hardware failures. The model exhibits such errors nondeterministically, since on the given level of abstraction it is not possible to express a definite cause for such errors. The ASM rule in Fig. 5 simply chooses an error code err' that complies with the predicate `create-pre` and only continues in case of success, i.e., the precondition is satisfied and no hardware failure occurs.

We enforce the strong guarantee that an unsuccessful operation must not modify the state in any observable manner. It takes quite some effort to ensure this behavior in the implementation.

Our POSIX model takes a lot of inspiration from previous efforts to formalize file systems. An initial pen-and-paper specification that is quite complete is [28]. Mechanized models include [1,17,25,19,9,21]. These have varying degrees of abstraction and typically focus on specific aspects. Both [17] and [21] are path-based. The latter proves a refinement to a pointer representation, file content is treated as atomic data. Model [9] is based on a representation with parent pointers (compare Sec. 3.2). They prove multiple refinements and consider power-loss events abstractly. The efforts [1] and [25] focus on an analysis of reads and writes with pages, in the first, data is accessed a byte at a time, whereas [25] provides a POSIX-style interface for read/write while integrating some effect of power-loss. An in-depth technical comparison to our model can be found in [15].

3.2 VFS + AFS as a refinement of POSIX

We have already motivated the purpose of the VFS and briefly described the integration with the AFS specification in Sec. 2 and [14]. We have proven our VFS correct wrt. the POSIX model relative to AFS [15]. As a consequence, any AFS-compliant file system can be plugged in to yield a correct system.

Conceptually, the VFS breaks down high-level operations to calls of operations of AFS. There are two major conceptual differences to the POSIX model:

1. The file system is represented as a pointer structure in AFS (as shown in Fig. 6) instead of an algebraic tree (compare Fig. 4), and
2. the content of files is represented as a sparse array of pages instead of a linear sequence of bytes.

These two aspects make the refinement proof challenging.

In VFS, directories and files are identified by “inode numbers” $ino : Ino$. The state of the AFS model is given by two (partial) functions $dirs : Ino \nrightarrow Dir$ and

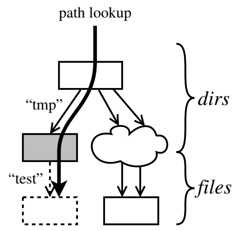

Fig. 6. FS as pointer structure

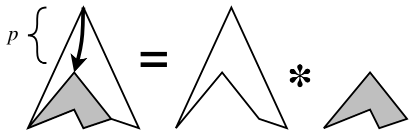

Fig. 7. Unfolding the tree abstraction, formally $\texttt{tree}(t, \mathit{ino}) = \texttt{tree}|_p(t, \mathit{ino}, \mathit{ino}') * \texttt{tree}(t[p], \mathit{ino}')$.

files : $\mathit{Ino} \nrightarrow \mathit{File}$. Directories *Dir* store a mapping from names of entries to inode numbers. Files *File* store a partial mapping from page numbers to pages, which are byte-arrays. The refinement proof establishes a correspondence between the tree t and *dirs* and between *fs* and *files* and maintains it across operations.

The first issue can be solved elegantly using Separation Logic [36]. The main idea of Separation Logic is that disjointness of several parts of a pointer structure is captured in the syntax of formulas. The main connective is the separating conjunction $(\varphi * \psi)(h)$, which asserts that the two assertions φ and ψ on the structure of the heap h are satisfied in *disjoint* parts of h.

In our case, the heap is given by the directories of the file system *dirs*, and the abstraction to the algebraic tree is an assertion $\texttt{tree}(t, \mathit{ino})$. It specifies that *ino* is the root of the pointer structure represented by t. The abstraction directly excludes cycles and sharing ("aliasing") in *dirs*, and establishes that a modification at path p is *local*: directories located at other paths q are unaffected.

To perform an operation at path p the abstraction `tree` is unfolded as shown in Fig. 7. The directory tree *dirs* is split into two disjoint parts, the subtree $\texttt{tree}(t[p], \mathit{ino}')$ located at path p and the remainder $\texttt{tree}|_p(t, \mathit{ino}, \mathit{ino}')$, which results from cutting out the subtree at path p, with root ino'. A modification in the grey subtree is then guaranteed to leave the white subtree unaffected.

The second issue manifests itself in the implementation of the POSIX operations `read` and `write`, which access multiple pages sequentially. There are several issues that complicate the proofs but need to be considered: There may be gaps in the file, which implicitly represent zero-bytes, and the last page may exceed the file size. The first and the last page affected by a write are modified only partially, hence they have to be loaded first. Furthermore, writes may extend the file size or may even start beyond the end of a file.

Since reading and writing is done in a loop, the main challenge for the refinement proof is to come up with a good loop invariant. The trouble is that the high number of cases we have just outlined tends to square in the proof, if one considers the intermediate hypothetical size of the file. We have solved this problem by an abstraction that neglects the size of the file, namely, the content is mapped to an infinite stream of bytes by extension with zeroes at the end. The loop invariant for writing states that this stream can be decomposed into parts of the initial file at the beginning and at the end, with data from the input buffer in between.

3.3 From AFS to LogFS to study Power Cuts

The UBIFS file system is *log-structured*, which means that there is an area on flash—the log—where all file system updates are written to in sequential order. Data in the log has not been integrated into the index that is stored on flash.

In the AFS model, there is no distinction between volatile state stored in RAM and persistent state stored on flash. Purpose of the LogFS model is to introduce this distinction on a very abstract level in order to study the effects of unexpected power cuts without getting lost in technical detail. The LogFS model thus bridges the gap between AFS and our FFS implementation.

The model simply assumes that there are two copies of the state, a current in-memory version *ramstate* and a possibly outdated flash version *flashstate*. The *log* captures the difference between the two states as a list of log entries.

Operations update the state in RAM and append entries to the log e.g., "delete file referenced by *ino*" or "write a page of data". The *flashstate* is not changed by operations. Instead, from time to time (when the log is big enough) an internal *commit* operation is executed by the file system, which overwrites the *flashstate* with *ramstate*, and resets the log to empty.

The effect of a power failure and subsequent recovery can now be studied: the crash resets the *ramstate*, recovery has to rebuild it starting from *flashstate* by recovering the entries of *log*. For crash safety we therefore have to preserve the recovery invariant

$$\mathit{ramstate} = \texttt{recover}(\mathit{flashstate}, \mathit{log}) \tag{1}$$

Note that although `recover` is shown here as a function, it is really a (rather complex) ASM rule in the model, implying that the invariant is not a predicate logic formula (but one of wp-calculus).

To propagate the invariant through operations it is necessary to prove that running the recovery algorithm on entries produced by one operation has the same effect on *ramstate* than the operation itself. This is non-trivial, since typical operations produce several entries and the intermediate *ramstate* after recovering a single entry is typically not a consistent state.

The possibility of orphans (Sec. 3.1) complicates recovery. Assume 1) a file is opened and 2) subsequently unlinked becoming an orphan, i.e., it is not accessible from the directory tree any more. The content of the file then can still be read and written through the file handle. Therefore, the file contents may only be erased from flash memory after the file is closed. However, if a crash occurs the process accessing the file is terminated and the file handle disappears. The file system should consequently ensure that the file content is actually deleted during recovery. There are two possibilities: the corresponding "delete file" is in the log before the crash, then no special means have to be taken. However, if a commit operation has happened before the crash but after 2), then the log has been cleared and consequently this entry will *not* be found.

To prevent that such a file occupies space forever, our file system records such files in an explicit set of flash-orphans *forphans* stored on the flash medium during commit.

Recovery thus starts with a modified flash state, namely $\mathit{flashstate} \setminus \mathit{forphans}$, and produces a state that contains no orphans wrt. the previous RAM state (denoted by *rorphans*). Invariant (1) becomes

$$\mathit{ramstate} \setminus \mathit{rorphans} = \mathtt{recover}(\mathit{flashstate} \setminus \mathit{forphans}, \mathit{log}) \tag{2}$$

It is not correct to require that only the recorded orphans *forphans* have been deleted as a result of `recover`. Again referring to the example above, in the case that no commit has occurred since event 2), we have $\mathit{ino} \in \mathit{rorphans}$ but $\mathit{ino} \notin \mathit{forphans}$, but the file *ino* will be removed anyway, since the deletion entry is still in the log.

Technically, in order to prove (2), additional commutativity properties between deleting orphans and applying log entries must be shown. This is nontrivial, since several additional invariants are necessary: for example, that inode numbers $\mathit{ino} \in \mathit{forphans}$ are not referenced from the directory tree of the RAM state, and that there is no "create" entry for *ino* in the log. Furthermore, since removal of directories and files is handled in a uniform way, directories can actually be orphans, too.

In our initial work [39] we have proved the recovery invariant on a lower level of abstraction, namely the file system core, but without orphans. Adding them to the model has lead to a huge increase in complexity, hence we switched to a more abstract analysis of the problem. It is then possible (although there are some technical difficulties) to prove that the recovery invariant of LogFS propagates to the concrete implementation by matching individual recovery steps.

The verification of recovery of a log in the given sense has been addressed in related work only by [31], which is in fact based on our model [39]. Their goal is to explore points-free relational modeling and proofs. They consider a simplified version of recovery that does not consider orphans, and restricts the log to the deletion of files only (but not creation).

Finally, we like to point out that the recovery invariant (2) needs to hold in every *intermediate* step of an operation—a power cut could happen anytime, not just in between operations. A fine-grained analysis is therefore necessary, e.g., based on a small-step semantics of ASMs, which we define in [33].

3.4 Node Persistence & Write Buffering

The persistence layer provides access to (abstract) erase blocks containing a list of nodes for the FFS core and its subcomponents. Clients can append nodes to a block and allocate or deallocate a block. This already maps well to the flash specific operations, i.e., appending is implemented by writing sequentially and deallocation leads to a deferred erase.

The node encoding layer stores these nodes as byte arrays on the flash device with the help of a write buffer. The implementation also ensures that nodes either appear to be written entirely or not at all. The write buffer provides a page-sized buffer in RAM that caches writes to flash until an entire page can be written. This improves the space-efficiency of the file system, since multiple nodes can potentially fit in one page. However, as a consequence of the caching of writes, it is possible that some nodes disappear during a crash. The challenge is to find a suitable refinement theory that includes crashes and provides strong guarantees in their presence. Clients should also be able to enforce that the write buffer is written. This is accomplished by adding *padding nodes*, i.e., nodes that span the remainder of the page. These nodes are invisible to a client of the persistence layer. The refinement between the persistence and the node encoding layer switches from indices in a list to offsets in a byte array as their respective input and output. The padding nodes complicate this refinement, since the abstraction relation from a list of nodes to the byte array is no longer functional.

For the sequential writes it is necessary that the layer stores how far an erase block is already written. However, it is difficult to update such a data structure efficiently, since only out-of-place updates are possible. Therefore, updates are performed in RAM and written to flash only during a commit operation similarly to the index. The recovery after a power failure has to reconstruct a consistent data structure and consider that some updates were not written to flash.

We formalize the byte encoding very abstractly, i.e., by just stating that decoding after encoded yields the original data. In future work we will consider whether we can use specificiation languages such as [2,27] to concretize the flash layout. Other models [25,9] neither consider the encoding nor buffer writes.

3.5 Erase Block Management

A raw flash device supports reading, sequential writing [10,16] and erasure of a block. Only after erasure it is possible to reuse a block. However, blocks degrade physically with each erase, i.e., typically after 10^4-10^6 erases a block becomes unreliable. There are two complementary techniques that deal with this problem and thereby increase the reliability of the device: wear-leveling and bad block management. The former means that necessary erasures are spread among the entire device evenly by moving stale data. Thus, bad blocks occur later in the lifetime of a device. Bad block management tries to detect which blocks are unusable and excludes them from future operations.

Both techniques are implemented transparently in the erase block management (EBM) by a mapping from *logical* to *physical* blocks. Fig. 8 shows this in-RAM mapping as arrows from logical to physical blocks. The client can only access logical blocks and the EBM keeps the mapping and allocates, accesses or deallocates physical blocks as needed. Wear-leveling copies the contents of a physical block to a new block and updates the mapping. Bad blocks are simply excluded from future allocations.

Another feature facilitated by the mapping is that clients may request an *asynchronous* erasure of a block. This increases the performance of the file system, because erasing a physical block is a slow operation. If the erasure is deferred, the client can already reuse the *logical* block. The EBM just allocates a different physical block.

The difficulty for the implementation and verification stems from interactions of the following factors (explained in more detail in [32]):

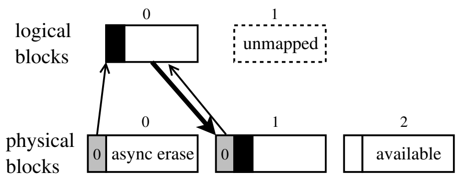

Fig. 8. Mapping in the EBM

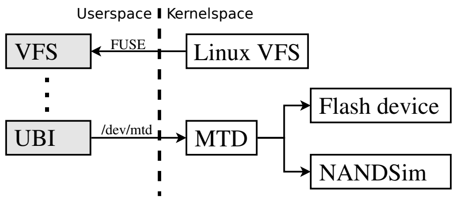

Fig. 9. Integration with Linux

1. The mapping is stored inversely within each *physical* block (shown as the backwards arrows in Fig. 8) and must be written before the actual contents.[1]
2. The flash device may exhibit nondeterministic errors. Only writing of a single page is assumed to be atomic.
3. Asynchronous erasure defers an update of the on-disk mapping.
4. Wear-leveling must appear to be atomic, otherwise a power-cut in between would lead to a state, where half-copied data is visible to the client.

Due to 1) and 3), the recovery from a power-cut will restore some older version of the mapping. This is visible to the client and has to be handled by the client. This necessitates a refinement theory that allows us to specify the behavior that a model exhibits in response to a power-cut and shows that an implementation meets this specified reset behavior, i.e., is reset-safe.

Items 1) and 2) complicate 4), because they lead to an updated on-disk mapping referring to half-copied data. Additional measures (such as checksums) are necessary to ensure that a mapping produced during wear-leveling becomes only valid once the entire contents are written.

Other formal models of EBM [25,26,9] usually do not take the limitation to sequential writes within a block into account, do not store the mapping on disk, assume additional reliable bits per block to validate the mapping only after the pages have been programmed or store an inefficient page-based mapping (assumed to be updatable atomically). Our model of the flash hardware is ONFI-compliant [16] and based on the Linux flash driver interface MTD [29]. The flash hardware model [6] is below our model of a flash driver.

4 Code Generation

We currently generate Scala [30] code, which runs on the Java virtual machine. We chose Scala, because it is object-oriented, supports immutable data types and pattern matching and features a comprehensive library. As explained below, we can generate code that is very similar to the KIV models. Therefore, Scala is well-suited for early debugging and testing of the models. Furthermore, it is easier to integrate a visualization for the data structures and operations. The

[1] The reasons for this are rather technical, but follow from the limitations of the flash hardware to sequential writes within a block and performance considerations.

final code is derived from the models shaded in gray in Fig. 2 and 3, but Scala code generation is also supported for the other models.

The code generation for Scala currently supports free and non-free data types (potentially mutually recursive). Most of the KIV library data types (natural numbers, strings, arrays, lists, stores and sets) and algebraic operations on them are translated to corresponding classes and functions of the Scala library. Furthermore, an algebraic operation with a non-recursive or structurally-recursive definition is transformed into a function that uses pattern matching to discern the cases of a data type.

We mimic the refinement hierarchy shown in Fig. 2 and 3 by inheritance, i.e., for each of the abstract (white) layers an interface is generated with one method for each of the ASM rules. For the concrete (gray) layers we generate an implementing class that has the ASM state as a member. The usage of an interface (—◎— in the figure) is implemented by aggregation. Thus, the VFS layer for example has an additional reference to an AFS interface. In the final file system this reference points to an instance of the FFS.

This setup also allows us to test the VFS model based on a different implementation of the AFS interface. We optionally also generate an implementation of the AFS interface based on the AFS ASM. This allows us to test a layer early on without having a refined model of all the used layers yet. These abstract layers however are rather nondeterministic, e.g., they heavily employ the following construct to choose an error code that may either indicate success or a low-level error (such as an I/O error).

$$\texttt{choose}\ err\ \texttt{with}\ err \in \{\texttt{ESUCCESS}, \texttt{EIO}, \ldots\}\ \texttt{in}\ \{\ \ldots\ \}$$

We currently simulate this by randomly choosing an element of the type and checking whether the element satisfies the given property.[2] This is sufficient when testing whether, e.g., VFS can handle the full range of behavior admitted by AFS.

However, in order to test whether a concrete model exhibits only behavior permitted by the abstract model this is insufficient. In that case it is necessary that the abstract model simulates the choices made by the concrete model. In future work, we will attempt to automatically derive an abstract model which is suitable for such tests. Each of the operations of this model may also take as input the output of the concrete layer and their goal is to construct a matching abstract execution. For the above example of error codes it is sufficient to just rewrite the abstract program by reusing the error code returned by the concrete model and checking that it satisfies the condition $err \in \{\texttt{ESUCCESS}, \texttt{EIO}, \ldots\}$. For more elaborate nondeterminism an SMT solver could be employed to make the proper choice.

We are currently also working on the generation of C code and expect approximately 10.000 lines of code. The C code generation will be limited to the models shaded in gray. In future work, we will investigate whether tools for the

[2] Note that for some forms of nondeterminism we can and do provide a better, deterministic implementation.

automated verification of C, such as VCC [7], could be employed to show the refinement between our last models and the generated C code automatically.

We are unaware of other file systems where code was generated automatically or the final code was verified against the models. [9] provides systematic translation rules that need to be applied manually to derive Java code.

The code generated from the models can be plugged into Linux via the *Filesystem in Userspace* (= FUSE) library [40], which allows mounting of a file system implemented in an application in user space. All calls to the mount-point are then routed to the application by FUSE. This allows us to test and debug the file system more easily. The access to either a real flash device (we currently use a Cupid-Core Single-Board Computer with 256 MB Flash [8]) or to the simulator (NandSim) is realized via the Linux device `/dev/mtd`. Fig. 9 shows the entire setup.

5 Lessons Learned

In this section we report on the lessons we learned during the design and verification of the flash file system.

In our opinion, the file system challenge is interesting for the reason that a wide conceptual gap must be bridged. Abstractly, the file system is described as a tree indexed by paths, whereas the hardware interface is based on erase blocks and bytes. The strategies we have taken from the Linux VFS and UBIFS to map between these interfaces deal with many different concepts.

Capturing these concepts at the right degree of abstraction proved to be a major design challenge. Not all concepts that we have encountered have direct counterparts in the implementation. Here, we benefit from the refinement-based approach that supports well isolating difficult aspects of the verification in dedicated models. For example, the abstract log as specified in LogFS is actually encoded in the headers of nodes in UBIFS.

It proved beneficial that we have initially started with the core concepts of the UBIFS file system and derived an abstract model [39]. This model has served as an anchor-point to incrementally develop the rest of the model hierarchy.

Our experience is that models tend to change frequently. One reason for that is that requirements are clarified or assumptions are rectified. It also happens that some model is refactored, in order to e.g. maintain a new invariant. Such changes are typically non-local and propagate through the refinement hierarchy.

The prime reason for changes stems from the fact that there is a fragile balance between simple abstract specifications and the question which functional guarantees can be provided by the implementation. The technical cause lies in power cuts and hardware failures, which affect almost all layers and can leak through abstractions (nondeterministic errors in POSIX are one example). More specifically, one needs to determine to which extent failures can be hidden by the implementation, how this can be achieved, and which layer incorporates the concepts necessary to address a particular effect of a failure. For example, the flash file system core can not express partially written nodes, so a lower layer (namely, node encoding in Fig. 3) must be able to recognize and hide such partial nodes (e.g., by checksums). In general, dealing with power cuts and hardware failures has increased the complexity of the models and their verification by a signification amount.

In order to reduce the effort inherent in frequent changes, KIV provides an elaborate correctness management for proofs. The system tracks the dependencies of every proof, i.e., the axioms and theorems used in the proof. A proof becomes invalid only if a dependency has changed. Furthermore, KIV supports replaying of an old, invalid proof. User interactions are therefore mostly limited to parts of a proof affected by a change.

Similar to [3], we observed that strong abstraction capabilities of the used tools are essential. KIV supports arbitrary user-defined data types (given suitable axioms), which was for example exploited to abstract the pointer structure to an algebraic tree and to abstract the sparse pages of files to streams (see Sec. 3.2).

We found that for smaller invariants it would be useful to have stronger typing support in our tool, such as for example predicative subtypes [37]. This could be used for example to express the invariant that every erase block has a certain size in the type. Making such invariants implicit reduces the number of trivial preconditions for lemmas.

6 Conclusion & Open Challenges

We have given an overview over our approach to the development of a formally verified flash file system. We outlined our decomposition into a hierarchy of models, described some of the challenges posed in each of them and sketched the solutions we developed. We are currently finalizing the models and their correctness proofs for the various refinements, one rather challenging remains: verifying the implementation of indices by "wandering" B^+-trees, that are maintained in RAM, and are incrementally stored on flash.

Two major aspects remain future work:

We have not yet addressed concurrency. A realistic and efficient implementation, however, should perform garbage collection (in the flash file system model), erasure of physical blocks and wear-leveling (in the EBM model) concurrently in the background. In those instances, where the action is invisible to the client of the respective layer, the abstract behavior is trivial to specify. However, on top-level it is not clear which outcomes of two concurrent writes to overlapping regions are acceptable, and how to specify them (the POSIX standard does not give any constraint).

Another feature that is non-trivial to specify is the effect of caching when using the real VFS implementation. The difference between our non-caching VFS implementation and the one in Linux would be visible to the user if a power-cut occurs, as not all previous writes would have been persisted on flash. For two simpler caching related problems (the write buffer of Sec. 3.4 and the mapping of Sec. 3.5) we already successfully applied a refinement theory that incorporates power-cuts [33].

References

1. K. Arkoudas, K. Zee, V. Kuncak, and M.C. Rinard. On verifying a file system implementation. In *Proc. of ICFEM*, pages 373–390, 2004.
2. G. Back. DataScript - A Specification and Scripting Language for Binary Data. In *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 66–77. Springer Berlin Heidelberg, 2002.
3. C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Lessons Learned From Microkernel Verification – Specification is the New Bottleneck. In *SSV*, pages 18–32, 2012.
4. E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15(1–2):237–257, 2003.
5. E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis.* Springer, 2003.
6. A. Butterfield and J. Woodcock. Formalising Flash Memory: First Steps. *IEEE Int. Conf. on Engineering of Complex Computer Systems*, 0:251–260, 2007.
7. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer Berlin Heidelberg, 2009.
8. `http://www.garz-fricke.com/cupid-core_de.html`.
9. K. Damchoom. An incremental refinement approach to a development of a flash-based file system in Event-B. October 2010.
10. Samsung Electronics. Page program addressing for MLC NAND application note, 2009. `http://www.samsung.com`.
11. G. Ernst, J. Pfähler, and G. Schellhorn. Web presentation of the Flash Filesystem. `https://swt.informatik.uni-augsburg.de/swt/projects/flash.html`, 2014.
12. G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV - Overview and VerifyThis Competition. *Software Tools for Technology Transfer*, pages 1–18, 2014.
13. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular Refinement for Submachines of ASMs. In *Proc. of ABZ 2014*, volume 8477 of *LNCS*, pages 188–203. Springer, 2014.
14. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. A Formal Model of a Virtual Filesystem Switch. In *Proc. of Software and Systems Modeling (SSV)*, pages 33–45, 2012.
15. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a Virtual Filesystem Switch. In *Proc. of Verified Software: Theories, Tools, Experiments*, volume 8164, pages 242–261. Springer, 2014.
16. Intel Corporation et al. *Open NAND Flash Interface Specification*, June 2013. URL: `www.onfi.org`.
17. M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying Intel flash file system core specification. In *Modelling and Analysis in VDM: Proc. of the fourth VDM/Overture Workshop*, pages 54–71. School of Computing Science, Newcastle University, 2008. Technical Report CS-TR-1099.
18. 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 Engineering of Complex Computer Systems*, 2008.

19. L. Freitas, J. Woodcock, and Z. Fu. Posix file store in Z/Eves: An experiment in the verified software repository. *Sci. of Comp. Programming*, 74(4):238–257, 2009.
20. T. Gleixner, F. Haverkamp, and A. Bityutskiy. UBI - Unsorted Block Images. `http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf`, 2006.
21. W.H. Hesselink and M.I. Lali. Formalizing a hierarchical file system. *Formal Aspects of Computing*, 24(1):27–44, 2012.
22. C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
23. A. Hunter. A brief introduction to the design of UBIFS. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`, 2008.
24. R. Joshi and G.J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2), June 2007.
25. E. Kang and D. Jackson. Formal Modelling and Analysis of a Flash Filesystem in Alloy. In *Proc. of ABZ*, pages 294–308. Springer, 2008.
26. E. Kang and D. Jackson. Designing and Analyzing a Flash File System with Alloy. *Int. J. Software and Informatics*, 3(2-3):129–148, 2009.
27. P.J. McCann and S. Chandra. Packet Types: Abstract Specification of Network Protocol Messages. *SIGCOMM Comp. Comm. Rev.*, 30(4):321–333, 2000.
28. C. Morgan and B. Sufrin. Specification of the unix filing system. In *Specification case studies*, pages 91–140. Prentice Hall Ltd., Hertfordshire, UK, 1987.
29. Memory Technology Device (MTD) and Unsorted Block Images (UBI) Subsystem of Linux. `http://www.linux-mtd.infradead.org/index.html`.
30. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.
31. J.N. Oliveira and M.A. Ferreira. Alloy Meets the Algebra of Programming: A Case Study. *Software Engineering, IEEE Transactions on*, 39(3):305–326, March 2013.
32. J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Formal Specification of an Erase Block Management Layer for Flash Memory. In *Hardware and Software: Verification and Testing*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013.
33. J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Crash-Safe Refinement for a Verified Flash File System. Technical report, University of Augsburg, 2014.
34. G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference*, pages 4186–4199. IEEE Computer Society, 2005.
35. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II, pages 13–39. Kluwer, Dordrecht, 1998.
36. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS*, pages 55–74. IEEE Computer Society, 2002.
37. J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.
38. G. Schellhorn. Completeness of Fair ASM Refinement. *Science of Computer Programming, Elsevier*, 76, issue 9:756 – 773, 2009.
39. A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract Specification of the UBIFS File System for Flash Memory. In *Proceedings of FM 2009: Formal Methods*, pages 190–206. Springer LNCS 5850, 2009.
40. M. Szeredi. File system in user space. `http://fuse.sourceforge.net`.
41. The Open Group. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2008 Edition. `http://www.unix.org/version3/online.html` (login required).