Delft University of Technology
Software Engineering Research Group
Technical Report Series

RAFFS: Model Checking a Robust Abstract Flash File Store

Paul Taverne and C. (Kees) Pronk

Report TUD-SERG-2009-033

TUD-SERG-2009-033

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of the 11th International Conference on Formal Engineering Methods, ICFEM 2009.

RAFFS: Model Checking a Robust Abstract Flash File Store

Paul Taverne and C. (Kees) Pronk

Delft University of Technology, The Netherlands
paultaverne@gmail.com, c.pronk@tudelft.nl

Abstract. This paper presents a case study in modeling and verifying a POSIX-like file store for Flash memory. This work fits in the context of Hoare's verification challenge and, in particular, Joshi and Holzmann's mini-challenge to build a verifiable file store. We have designed a simple robust file store and implemented it in the form of a Promela model. A test harness is used to exercise the file store in a number of ways. Model checking technology has been extensively used to verify the correctness of our implementation. A distinguishing feature of our approach is the (bounded) exhaustive verification of power loss recovery.

1 Introduction

A software product should meet all of its requirements and fully conform to its specification. Testing and other quality assessment techniques can only help to approach this goal [9]. Proving that a piece of software will always work correctly requires the use of formal methods. Two approaches to using formal methods exist: *Post facto* and *correctness by construction* [29]. The first approach uses technologies like model checking to determine the correctness of (an abstraction of) an existing implementation. This is often called *verification*. The second approach can be used to construct an implementation starting from an abstract formal specification. This technique is sometimes referred to as *refinement*.

Tony Hoare has proposed a Grand Challenge project [17], whose long-term vision is to develop methodologies and a set of automated tools that can be used to verify whether a piece of software meets its requirements [5]. One of the steps towards the realization of this vision is to build a repository [40] of formalized software designs and verified implementations, which can be used to test and develop the before mentioned tools.

The first case study for the Verification Grand Challenge was Mondex [40], a smartcard functioning as an electronic purse. At the VSTTE conference [41] in Zürich in 2005, Gerard Holzmann proposed a *mini-challenge*: build a verifiable file store specifically designed to work with Flash memory [30, 23]. The mini-challenge defines strict robustness requirements. The file store should be able to cope with unexpected power loss without getting corrupted. It should also be able to recover from faults specific to Flash memory, such as bad blocks and bit corruption. The mini-challenge has been embraced by the Formal Methods

community. It has for example been a case study for the ABZ conference [2, 1] and has also been a topic at several VSR-net workshops. Precursors of this paper have been presented at several GC6 workshops [14].

This paper presents our work contributing to the mini-challenge project. This work is experimental in nature. Our approach is unique in that it forms a kind of middle road between abstracting an existing implementation and the correctness by construction approach. We have designed a simple file store called RAFFS, which is short for **R**obust **A**bstract **F**lash **F**ile **S**tore. This file store and its surrounding environment has been implemented in the modeling language Promela [18]. The environment includes a Flash memory and a test harness that interacts with the file store. RAFFS is capable of fixing inconsistencies that may be present in the file store after a power failure. Our model includes the injection of such power failures. For simplicity reasons we have assumed that the Flash memory is fully reliable. Bit flips and bad blocks are thus not included in the model.

A 'proof of correctness' for our file store implementation is obtained by (exhaustive, but bounded) model checking. At the end of the paper we present some measurements relating model checking particulars (such as memory usage and running time) to the 'depth' of testing by the test harness.

In Section 2 we give an overview of related work and in Section 3 we discuss the quirks of Flash memory. Section 4 will discuss abstractions and simplifications that we have applied to our model. Section 5 will give some implementation details, Section 6 will give verification results. Finally, Section 7 will provide conclusions and suggestions for future work.

2 Related work

File systems are abundant in computer systems, however the correctness of their implementations is seldom proved. We will first discuss several papers based upon the refinement approach.

A number of authors have used refinement approaches [4], [6], [7], [10], [11], [25], [31] and [33]. Of particular note, Morgan and Sufrin [33] give a specification of the Linux file system using the Z-notation [38]. They explicitly abstract from issues such as data representation and the physics of the underlying storage medium. Their specification is one-level only; there are no refinement steps towards an implementation. Arkoudas et al. [4] prove an implementation of a file system correct by establishing a simulation relation between a specification of the file system and an implementation. The specification models the file system as an abstract map from file names to sequences of bytes. In the implementation, fixed sized blocks are used to store the contents of the files. The implementation assumes an ideal storage medium. Their proofs use the Athena system [3] and automatic theorem provers. Kang and Jackson [31] use Alloy [28] to construct a formal model of a Flash file system. Their model includes the underlying hardware and the file system software with basic operations such as read and write.

A fault tolerance scheme addresses memory issues such as the management of block erasures, wear leveling and garbage collection.

As examples of the verification approach, we mention two important papers. Galloway et al. [13] use model checking to investigate the correctness of a Linux Virtual File System (VFS). They downscale existing VFS code by slicing away code and abstracting data. They transform C-code by hand into Promela and SMART models. This turns out to be a challenging task, partly because of the shallow VFS documentation. They had to reduce the sizes of their file system data structures (such as inodes) to similar values as we have used during the construction of our RAFFS. However, due to the well known state explosion problem, they seem unable to perform exhaustive model checking, whereas we have been able to apply exhaustive model checking widely. Yang et al. give an important start to the verification of a file system using model checking [42]. Like in our approach, Yang et al. check file systems for storage errors and they use model checking to verify that a file store, upon encountering such errors, will reboot into a known legal state. They operate the OS and several known file stores from within the model checker whereas we have separated out the file store. The article by Mühlberg and Lüttgen [34] is also an example of exhaustive testing of existing code using model checking. They use BLAST [16] to check device drivers for memory safety (illegal pointer de-references) and locking behaviour.

We have created a POSIX-like file store using a Flash memory. Our file system includes files, directories, reads and writes, block erasure and garbage collection. Since we have assumed the Flash memory to be reliable, we did not model wear, bad blocks, and bit corruption. A distinguishing feature of our model is that it includes the simulation of general system failure in the form of power loss. Our file store has been designed to always recover to a consistent state. Where others have mainly focused on the verification of specifications, we have focused on verification of the implementation. Our implementation supports multiple simultaneous users of the file store.

3 Flash in a nutshell

For compatibility with existing operating systems, it is desirable that a Flash memory acts as a block based device. Most Flash device drivers emulate a block based device. All Flash specific behavior is then handled in a special layer called the Flash Translation Layer (FTL) [27].

Flash memory has a hierarchical structure similar to common block based storage devices, but has some restrictions with regard to its usage. The memory is divided in small chunks called pages. Pages are grouped together into blocks. A page is the smallest access unit for reading and writing. Writing to a Flash page, which is called programming, has an important limitation. It is only possible to change bit values from 1 to 0. In an empty (read: erased) page all bits have value 1. An empty page can be programmed with any desired data. But once a page has been programmed, it can not be overwritten with arbitrary data. It must first be erased, resetting all bits back to value 1. Erasing is a special operation

for Flash memory. The smallest unit for erasing is a block, making it impossible to erase individual pages.

Consequences of the above mentioned limitations are that page content can not be overwritten and that pages with obsolete content can not be erased immediately. The first issue is solved by performing *out of place updates*. This means that instead of overwriting data in its existing location, the data is written to a different (free) location. The original data is marked as obsolete. A garbage collection algorithm is responsible for erasing blocks that contain pages with obsolete content. Valid content can be moved elsewhere when needed.

Moving data around puts a new problem on the table. The file store must know where every single piece of data is stored. This information is part of the metadata stored in for example inodes. Updating an inode whenever a data address changes would result in a vicious circle. This problem is solved by adding a level of indirection between the translation of logical addresses to physical addresses. The indirection and the moving of the data is handled by the FTL. From the point of view of the file store data can then be overwritten in its current logical location and addresses stored in inodes do not need to be updated. The FTL maintains a mapping table in RAM to translate logical addresses into physical ones. The logical address that is associated with the contents of a Flash page is stored in a special part of the page called the *spare data region* along with other metadata used by the FTL. The mapping table can be rebuild by reading this metadata from Flash (during mounting).

Flash memory is well known to have some reliability problems. For example bits may randomly flip value due to electrical interference in the memory. Deterioration of the material of which the memory is made can lead to damaged blocks. We have excluded these issues from our model and have assumed a perfectly reliable Flash memory where no data corruption or unexpected data loss may occur. Properly dealing with these reliability issues is a very complex task and depends heavily on statistics. Our Promela model of a downscaled file store is not intended to be used for statistical analysis. Instead we use exhaustive verification to prove complete correctness.

4 Abstractions

4.1 POSIX Compatibility

The mini-challenge project suggests using a subset of the POSIX standard [35, 36, 12]. Our highly abstracted file store API is not fully POSIX compatible. We will compare our API with the abstract formal specification [33] from Morgan and Sufrin. An overview of the file store API functions in RAFFS is given in Figure 1.

There are some differences between our API and the formal specification. In the formal specification, a directory is encoded and stored as a file, and all API functions operate on files. In our model, a clear distinction is made between files and directories, and there are separate functions for dealing with these objects.

This design choice simplifies the implementation and downscaling of the file store. For example, a directory inode can now be stored in a single page, instead of multiple.

Our model does not contain *file descriptors*, or *channels* as they are called in the formal specification. We have abstracted those away to reduce memory usage. Thus the functions *open*, *close*, and *seek* are not present in our model. Instead of a file descriptor, all functions in our API have a parameter *path* which is used to uniquely identify the inode on which the function operates.

In our model, each inode always has a name. There is no separate naming system like in the formal specification. As a consequence, our model lacks the *link* and *unlink* functions. We have added two delete functions for removing files and directories.

The handling of data also differs from the formal specification. Our API functions only have a single unit of data as input or output, instead of a sequence of data units. The reason for this abstraction is to reduce the complexity of the file store implementation. Changing this, possibly as part of future work, will not have an impact on the robustness quality of RAFFS.

The formal specification lacks clear definitions of error conditions. We have clearly defined error conditions for all API functions. Our implementation checks for those conditions and returns appropriate error codes.

```
fs_api_create_file(path);
fs_api_file_exists(path);
fs_api_file_size(path);
fs_api_file_append(path, data);
fs_api_file_modify(path, offset, data);
fs_api_file_read(path, offset);
fs_api_file_truncate(path, size);
fs_api_delete_file(path);
fs_api_create_dir(path);
fs_api_dir_exists(path);
fs_api_dir_size(path);
fs_api_dir_get_child_name(path, index);
fs_api_delete_dir(path);
fs_api_mount();
fs_api_unmount();
```

Fig. 1. File store API function prototypes

4.2 Abstractions applied

The goal of our project is to perform (exhaustive) verification of our state-based model, which requires that the model must have a low complexity. In the current context, complexity means the size of the state space. The bigger the state space, the more memory and time is needed to perform verification.

Abstracting an existing implementation would be a time consuming task; far more work than we could accomplish in a short period of time. Others abstracted

parts of an existing file store [13] which proved to be a difficult task. We have decided to design a basic file store from scratch. So instead of abstracting a complex design, we have directly made an abstracted design. The advantage of this method is that we had complete freedom in the design choices, allowing us to keep things simple, while also making a robust design. We were not forced to think within the paradigm of other designers. We are confident that this choice has resulted in a model with a much lower complexity than could have achieved if we would have attempted to abstract (and modify) an existing implementation. A disadvantage of making a whole new design and implementation is of course that it is difficult to compare our work with other designs and implementations. Making the design more similar to real-life file stores is envisioned as future work. Our current efforts should be seen as a demonstration of the capabilities of model checking and the complexities involved with verifying a file store implementation.

There are many abstractions and simplifications in our model. The simplified API and the assumption that the Flash memory is reliable have already been discussed. Another simplification is related to inodes. In the model there are two distinct types of inodes, namely a file and a directory. A file contains a sequence of data units. A directory contains an unordered list of references to child inodes. Both types of inodes are assumed to be able to always fit into a single page. They will thus never span multiple pages. The data of a directory is not encoded as file data and is not stored as a file. When a page is used to store file data instead of an inode, then we assume that a page can fit exactly X units of data, regardless of how big a page really is. A unit of data in the model is a single bit.

There are different types of data abstractions in the model. The first type of abstraction is the compact representation of data. For example, file names and paths are not represented by character sequences, but by numerical values. These numerical values can easily be stored as short sequences of bits. The values themselves are not really important in an abstract model. We only need a certain number of distinct values.

The second type of abstraction is to limit ranges of values. Everything in the model is scaled down to a small size. Sizes in the model are specified by constant values. This allows us to easily modify those sizes when desired. The values of these constants have been carefully chosen to scale down the file store as far as possible without sacrificing functionality. For example, every single error condition in each API function must be reachable.

The flash memory in the model currently consists of just 4 blocks, each having two pages, giving a total of 8 pages. The maximum number of inodes that can be present in the file store is currently set to 4. The maximum size of a file is 3 data units. The maximum number of children of a directory is 2.

5 Implementation

In the sections below we will discuss the model that we have constructed. This discussion will necessarily be limited. Further details can be obtained from [39]. Our model has a layered design, consisting of five layers as shown in Figure

2. The arrows indicate which layer uses functionality from another layer. The bottom two layers are the Flash memory and its driver, to be discussed in Section 5.1. The next two layers belong to our file store implementation, named RAFFS, which is discussed in Section 5.2. The top layer serves as a test harness for RAFFS. This test harness is discussed in Section 5.3. In Section 5.4 we explain how power loss has been modeled. Section 5.5 discusses how the design of RAFFS has been adapted to meet the robustness requirement. Finally, Section 5.6 explains the issues of verifying a model that contains multiple users.

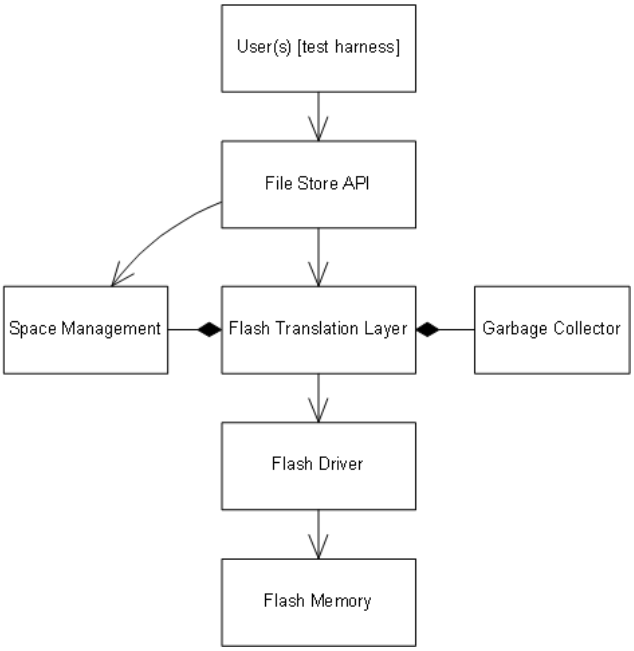

Fig. 2. UML layer diagram

5.1 Flash Memory and its Driver

The bottom layer in our implementation is the Flash memory. We have modeled the Flash as a simple data structure, namely an array of pages. Each page consists of two parts, the data region and the spare data region. The data region is used for storing inode metadata or file data. Its size is chosen so that it is exactly large enough to fit the largest possible inode in the model. The spare data region is used for storing metadata used by the file store, and the Flash Translation Layer in particular. This metadata includes three page status bits, a version field, and a virtual page number.

The second layer in the model is the Flash driver, which has functions to read or program a page, and to erase a block. Details of both layers are shown in Figure 3.

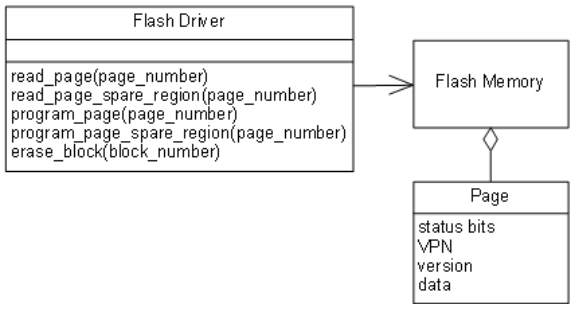

Fig. 3. Flash Memory and Driver

5.2 RAFFS

The third layer is the Flash Translation Layer (FTL), which implements Flash specific behavior and emulates a generic block based storage device. It exposes functions to the file store API layer for writing, updating, and deleting data in a logical location. The FTL maps logical addresses to physical ones and performs *out of place updates* as explained in Section 3. We use a simple mapping scheme based on virtual page numbers (VPN) [8]. Other schemes exist that are more memory efficient because they use a smaller mapping table [32]. However, due to the extremely small scale of our Flash memory, implementing such a complex scheme would require increasing the number of Flash blocks. Using a different scheme is therefore not worthwhile for us in terms of both complexity and memory efficiency.

The FTL applies a two step programming protocol [26] when writing data to a Flash page. This protocol is a required for robustness reasons.

The FTL contains a simple garbage collection algorithm for recovering space occupied by pages with obsolete content. The FTL in our model is also responsible for the management of free space. To ensure that enough free space is available to successfully complete an operation, each API function must reserve an amount of free locations that equals the number of writes that it is planning to perform.

The UML diagram in Figure 4 shows the functions that the FTL exposes to the file store API layer. The fourth layer is the file store API, which was already discussed in Section 4.1. A UML diagram with details of this layer is shown in Figure 5.

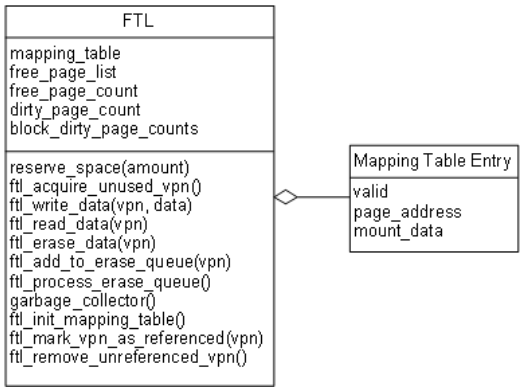

Fig. 4. Flash Translation Layer

5.3 Test Harness

The fifth and topmost layer is the user layer, which functions as a test harness in our model. A user is a Promela process that calls file store API functions. Multiple concurrent user processes are allowed to be present in our model. We use simple techniques to prevent processes from interfering with each others' operations. Shared data structures are either locked for exclusive access or values are read and updated within a single atomic block. Locking is for example applied to inodes.

We have made several variations of our model, each with a different testing purpose, and with a different implementation of the user layer. Four variants will be discussed in this paper; *SU* (single user), *SUPL* (single user with power loss), *MU* (multiple users), and *MUPL* (multiple users with power loss). The SU variant is discussed below, the SUPL variant is discussed in Section 5.4, and the MU and MUPL variants are discussed in Section 5.6.

To verify the correctness of the file store, we must consider all possible states in which the file store can reside, and all possible transitions between those states. The *SU* variant of our model contains a single user process that performs a random sequence of file store API calls. In each step of the sequence, the file store API function and its inputs are chosen non-deterministically. Performing full verification on this model will, with an infinite sequence, examine the entire state space.

The behavior of every file store API call should be exactly as described in the specification. We verify this by comparing the result of each API call with the result given by a reference implementation using assertions. The idea is that incorrect behavior will always lead to an unexpected result being returned at some point in the sequence. Because of the state-based nature of our model, the reference implementation has to be fully integrated in the code. If the model would be used for simulation only, then an external reference implementation could be

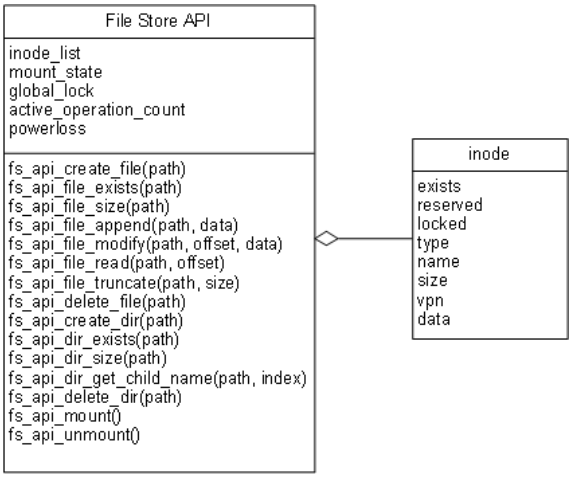

Fig. 5. File Store API

used. Holzmann used an external reference implementation during randomized differential testing of his own Flash file store model [15].

Because the reference implementation needs to be integrated into our model and because our file store API is unique, we were forced to make our own reference implementation. The reference implementation has thus also been written in Promela. This implementation has not been verified to be correct, we can only assume it is correct. Proving its correctness is a task for future work. Suppose that one of the two implementations contains a bug. This bug would only go unnoticed only if both implementations would exhibit the exact same incorrect behavior. However, since the two implementations are considerably different such a situation is unlikely.

The reference implementation in our model is an extremely abstracted implementation of a file store. It maintains a private list of inodes that are expected to exist in the file store. All relevant data is stored in those inodes, including file contents. The file store API functions operate directly on the inode list. The reference implementation is much less complex compared to our RAFFS implementation, as there is no storage medium, and no Flash specific behavior. The code size of the reference implementation is an order of magnitude smaller than the RAFFS implementation.

The sequences of API calls that are performed by the test harness have a configurable length. This allows us to control the 'depth' of our verification. Increasing the sequence length will increase the size of the state space. An unbounded model would be far too complex for performing exhaustive verification. The impact of the bound will be discussed in Section 6.2.

As said before, the input parameters for the chosen API function are generated non-deterministically. The generated values of the input parameters fall in a domain of potentially valid input values. This means values which, in certain

situations, could lead to a successful file store operation, while in other situations they will trigger an error condition in the file store API function. Values that always trigger an error fall outside of the inspected domain. We have tested such 'invalid' inputs in a separate deterministic test suite.

Besides comparing results with a reference implementation, we have also implemented a series of consistency checks which are performed after each file store API call. These consistency checks compare metadata stored in RAM by the file store with metadata stored on the Flash memory using assertions. Inodes and the mapping table of the FTL are examples of such metadata.

5.4 Power loss

One of the goals in our project is to make the file store robust so that it can cope with power failures. If power loss occurs while there are unfinished file store operations, the contents of the Flash memory may be in an inconsistent state. The mount operation is responsible for detecting inconsistencies and recovering the system to a valid consistent state. The robustness requirement specifies that an unfinished operation must either be completed, or that all changes made by such an unfinished operation must be undone.

A second variant of our model, named *SUPL*, is an extension of the SU variant discussed in the previous section. The main difference is the addition of power loss in the model. Promela has a special language construct called *unless* that we have used in this variant. The *unless* construct works similar to an exception handler. It contains a main sequence of statements and an escape sequence. Before a statement from the main sequence is executed, an escape condition is checked. If this condition evaluates to true, execution jumps to the escape sequence. The remaining statements in the main sequence are not executed.

The code that we want to be susceptible to power loss is put inside an *unless* construct. A second Promela process is used to trigger power loss. When power loss occurs, execution of the file store code is aborted. All data that the file store maintains in RAM is cleared and the file store is re-mounted. During mounting the contents of the Flash memory is checked for consistency and corrections are made. After mounting a new sequence of API calls is performed. In our model, multiple subsequent power loss situations take place. They can occur at any time, even during mounting.

Figure 6 shows pseudo code for the model variant SUPL. From the point of view of the user there can be two valid situations when power loss has occurred during an operation. One in which the operation was completed successfully, and one in which the operation was not performed at all. We therefore let the reference implementation maintain two states. Each state consists of an inode list and the amount of available free space. The first state is the 'before' state, where the chosen API function has not been performed. The second state is the 'after' state, where the chosen API function was performed successfully by the reference implementation. If a power loss situation occurs, then after mounting we compare the state of the file store from the RAFFS implementation with the two states from the reference implementation and pick the first one that matches

(function *select_expected_state* in the pseudo code). The reference implementation will continue with that state as both the new after and before state. If neither state matches, then the robustness requirement was not met and there is a bug in the model. If the chosen API function completes without the occurrence power loss, then the return values of both implementations are compared (function *compare_results* in the pseudo code). The return value either is an error code (negative value), a data value (positive value), or a void (zero value). The states of the reference implementation are also synchronized, the before state being set equal to the after state.

The results presented in Section 6.2 were obtained using a model that triggers a fixed number of 2 power loss situations during each sequence. Triggering more power losses, for example by using an infinite loop, will increase the complexity of the model.

```
bool powerloss;

PROCESS_USER() {
  initialize_system;
  while(1) {
    {
      mount;
      select_expected_state;
      perform_consistency_checks;
      while(1) {
        choose_api_function;
        generate_inputs;
        perform_api_call_reference;
        perform_api_call;
        compare_results;
        perform_consistency_checks;
      }
    } unless {
      powerloss == 1
    }
    reboot;
  }
}

PROCESS_TRIGGER_POWERLOSS() {
  /* multiple power loss situations */
  powerloss = 1;
  powerloss = 1;
  ...
}
```

Fig. 6. Pseudo code for SUPL test harness

5.5 Mounting and order of operations

The mount operation reads inode metadata from Flash and stores it in RAM. It fills the FTL mapping table and is also responsible for correcting inconsistencies in the file store metadata residing on the Flash memory. Our solution for making

the file store robust involves making changes to the contents of the Flash memory in a specific order. Given that order, a small set of generic corrective actions will always result in fixing all inconsistencies. The resulting state will be one in which all unfinished operations have either been completed or been undone.

To explain the specific order in which Flash content modifications are made during the various file store operations, we first classify two types of operations. Operations that add inodes or modify data are classified as *constructive*. Operations that remove inodes or data are classified as *destructive*.

We make use of three relations between different pieces of information stored on the Flash memory to decide which corrective actions must be taken by the mount algorithm. Firstly, all inodes (except the root) are referenced by the inode of its parent directory. Secondly, every data page is referenced (through its VPN) by a file inode. Thirdly, when two pages have identical VPN value, the version field can be used to determine which one is the newest copy. If no relation between an inode can be found with another inode, then this inode is called an orphan. Unreferenced data pages are handled in a similar way. Orphaned entities will be removed during mounting. Items that become orphaned during the removal process will be removed as well.

For destructive operations, the idea is to create orphans as quickly as possible. The operation can then always be completed if aborted unexpectedly. For example, when removing a file, the very first step to take is updating the inode of its parent directory. The file inode and any related data, will then become orphaned.

For constructive operations, the idea is to keep all new and modified data orphaned as long as possible. Also, existing data must be kept intact. Then the changes made by the operation can always be undone if aborted. When creating a file, the first step is to create the inode of the file and the last step is to update the inode of its parent directory. The functions in the file store API should thus not overwrite file data. Instead they must write updated data to a new virtual location. The old data locations are deleted at the end of the operation (when they have become orphaned). By preserving the old data, a rollback is possible.

The four corrective actions that our mount algorithm makes are: (1) removing unreferenced inodes, (2) removing unreferenced data pages, (3) removing pages with content of which a newer version was found, (4) removing pages that have state invalid. Removing means marking as obsolete. The actual erasing is done later by the garbage collector. A page can have state invalid if power loss occurred during the first step of the two step programming protocol.

5.6 Multiple users

The verification method that we have described in the previous sections was performed with a single user process. Even though our implementation supports multiple users, it is not possible to apply the same method to a model with multiple simultaneous users. The reason is that the expected result of each API call is not predictable. We have two variants of our model that contain multiple users; variant *MU* without power loss injection, and variant *MUPL* with

power loss injection. In these variants each user again executes a sequence of file store API calls, but this time the results are not compared to a reference implementation. Only general consistency checks are performed. Our main focus with these variants has been on verifying the absence of deadlock. We hope to do more extensive verification in the future. Additional processes cause the state space to explode, making it currently impossible to perform any exhaustive model checking on the multi user variants, even with a bounded model.

6 Verification and Results

The model checking tool SPIN [22] was used for verification of our model. Results of our verification efforts are presented and discussed in Section 6.2. First we will present a method that we developed for compact storage of variables.

6.1 CCVS: Custom Compact Variable Storage

Our model contains many variables with a small size, typically of the special Promela variable type *unsigned*. SPIN maps such variables onto (larger) integer type variables (byte/short/int). As a result, the size of the state vector can be larger than strictly required. Inspired by the work of Ruys [37] we have implemented a generic method for storing the values of multiple variables into a single integer type variable, which then serves as a container. The purpose of this storage method is to reduce memory usage during verification. Bit arithmetic is used for reading and writing the values of individual variables from such a container variable. CCVS makes heavy use of preprocessor macros. We have currently defined variables both as normal individual variables and using our CCVS method. One of these two storage methods is chosen with a preprocessor flag. Disabling CCVS can be useful for debugging purposes, since it obfuscates the variables.

6.2 Results

We will first discuss results obtained for the *SU* variant that was described in Section 5.3. We have been able to perform exhaustive verification for sequences (of file store API calls) up to length 6 with a memory usage less than 1 GB. Using a machine equipped with a 2.3 GHz Intel Xeon CPU and 32 GB of RAM we were able to go up to length 8. Sequences of length 15 and 31 have been extensively tested using the bitstate hashing technique [20].

It is important to note that, thanks to the small scale of the file store, short sequences will already result in many interesting situations being tested. Based on experience gained by making a deterministic test suite, we can say that the majority of error conditions and corner cases in the file store are reachable at length 5 or below. Sequences with a length above 8 mainly involve exploring states that could be considered variations of states reachable at shorter lengths. Complete verification, with an unbounded sequence, is currently infeasible. Nevertheless,

the results that we have obtained give us confidence that our implementation functions as intended and expected. In the future we plan to extend the reachable part of the entire state space by starting with different initial states.

Sequence length	CCVS	States	Memory (MB)	Time (s)
2	no	129,835	21	0.2
2	yes	137,581	14	0.2
3	no	1,140,027	168	1.7
3	yes	1,217,514	105	1.7
4	no	9,332,851	1,387	14.5
4	yes	10,039,814	876	14.0

Table 1. Results for variant SU. Compression: none.

Results from our exhaustive verification runs can be found in Table 1, Table 2, and Table 3. Each table shows the number of states, the memory usage, and the execution time for different sequence lengths of file store API calls. The difference between the three tables is the state compression method that was chosen in SPIN. No compression was used for the results listed in Table 1, *state vector collapse* [21] was used for Table 2, and *minimized DFA encoding* [24] was used for Table 3.

Sequence length	CCVS	States	Memory (MB)	Time (s)
2	no	129,835	7	0.4
2	yes	137,581	7	0.3
3	no	1,140,027	43	3.6
3	yes	1,217,514	39	3.0
4	no	9,332,851	344	29.0
4	yes	10,039,814	319	24.4

Table 2. Results for variant SU. Compression: -DCOLLAPSE.

CCVS proved to be very useful for reducing memory usage during verification of our model. To our surprise it also consistently reduced the time needed to verify the model, despite the additional computations that it makes. The benefit of CCVS is particularly significant when using SPIN's minimized DFA encoding technique (Table 3), where it reduced memory usage by more than half. The results show that minimized DFA encoding gives the lowest memory usage, even without CCVS, at the cost of a significantly longer execution time. We have used this powerful compression technique, in combination with CCVS, during the rest of our verification efforts since memory is the most scarce resource available to us.

Sequence length	CCVS	States	Memory (MB)	Time (s)
2	no	129,835	6	3.4
2	yes	137,581	4	2.0
3	no	1,140,027	27	34.2
3	yes	1,217,514	14	19.3
4	no	9,332,851	147	317.0
4	yes	10,039,814	60	164.0
5	yes	67,763,329	249	1,170.0
6	yes	365,532,240	900	6,710.0
7	yes	1,801,435,400	5,989	33,800.0
8	yes	8,010,865,300	18,577	156,000.0

Table 3. Results for variant SU. Compression: -DMA.

The use of model checking proved particularly useful during the verification of the model variant SUPL that included power loss injection, which was discussed in section 5.4. By analyzing all possible execution interleavings of the processes in the model, we have been able to verify that our implementation is always able to recover to a valid consistent state, regardless of the moments at which power loss situations occur.

Sequence length	CCVS	States	Memory (MB)	Time (s)
2	yes	13,205,390	98	268.0
3	yes	93,392,252	882	2,040.0
4	yes	727,651,040	5,434	16,700.0
5	yes	5,467,663,300	28,584	134,000.0

Table 4. Results for variant SUPL. Compression: -DMA.

We have been able to perform exhaustive verification for a sequence length of 5 on a machine with 32 GB RAM. Results for verification runs on the SUPL variant are listed in Table 6.2. The applied state compression technique was very efficient with a compression of 95.4% at length 5. This means that without compression the verification run would have required 626 gigabyte of memory. Without CCVS it would have even been double that amount. The execution time of 37 hours is long but in our case more than acceptable. Being able to trade time for reduction in memory usage has allowed us to perform exhaustive verification for sequences longer than would otherwise have been possible.

We already started using the described verification methods during the development of our implementation. This has helped us to find flaws in our implementation quickly and effectively. Like a compiler can point out syntax errors in its input, a verifier can reveal errors in the functioning of its input. By analyzing and fixing our mistakes, we gained a better understanding of the functioning of

the model. This knowledge was beneficial for the remainder of the development cycle.

We found a few bugs in our implementation through model checking. In our garbage collection algorithm we found a bug that would probably have never been discovered without the help of model checking techniques. A few free pages are always implicitly reserved for garbage collection so that valid content can be moved from a dirty block to other blocks, before the dirty block is erased. It turned out that the situation in which a power failure occurs right before the erasure of a block was not properly handled. In that case the number of available free pages could decrease below the minimum amount needed to perform garbage collection on an arbitrary dirty block. This flaw was fixed by adjusting the garbage collection algorithm to process the dirtiest blocks first.

We have been unable to perform exhaustive verification on the MU and MUPL variants of our model. The state space explodes when multiple Promela processes are used in a model. Approximative verification has been applied on these variants using SPIN's bitstate hashing technique. Bitstate hashing explores only a subset of the entire state space. Based upon the results obtained and the complexity estimates given by Spin, it is difficult to estimate the full size of the state space. Results of this testing have therefore been omitted from this paper. Our focus up till now has been mainly in the single user variants. We plan to shift our focus to the multi user variants in the near future.

7 Conclusion and Further Work

We have presented the construction of a POSIX-like file store for Flash memory, specially designed for model checking. The system has been constructed using the Promela modeling language. The model abstracts from a real file store by reducing various data structures and by reducing the number of operations supported.

Distinguishing features of our model are the exhaustive verification of the code and the ability to inject multiple power losses in the functioning system. Upon recovery from power loss, the system status is restored into a known correct state. This mechanism has been exhaustively verified. Multiple user processes exercise the model to its full extent under exhaustive verification using a test harness. The paper gives quantitative results of the model checking process such as the compression mechanisms used, number of states investigated, memory usage and running time.

The exhaustive verification makes our approach stand out from other ways of verifying a file system. In [13] the code of a Linux VFS is downscaled and transformed into Spin and SMART models. Although the authors of that paper downscale to similar values as we have used, they seemed unable to use exhaustive verification on their model. Additionally our model verifies Flash particulars such as out of place updates and garbage collection. Our approach differs from [42] in that we test a special purpose file store for Flash memory whereas they concentrate upon existing file systems. Compared to various papers based upon

the refinement approach we have constructed real code, resulting in a functioning system.

The middle road between the above approaches we have followed by constructing an abstract model in Promela has proven to be very useful to test a design concept on a reduced scale. Our model proved to be truly effective in verifying the ability of the RAFFS implementation to recover from power loss.

We can't claim full verification of our implementation because of the bounds that we have set in the model to limit the size of the state space, and also because of the assumption that our reference implementation is correct. However, the results that we have obtained give us high confidence in the correctness of our implementation. Our efforts could be considered as an extreme form of testing.

RAFFS has no direct practical application since it is a highly abstracted design and implementation. That was also never our intention. Our work gives a good indication of the complexities involved with model checking a file store implementation. Even an abstracted and downscaled file store is too complex to perform unbounded exhaustive verification with currently available hardware and technologies. The inevitable dependency on data is one of the causes of the enormous size of the state space.

We envisage reducing some of the current severe abstractions in the model. A goal is to make our implementation more realistic and more comparable to existing implementations and specifications. We want to make the file store API more similar to POSIX. A desirable modification is to store directories as files like is done in UNIX. This will also require upscaling the file store. The page based mapping in the FTL can be replaced by a more complex (and more memory efficient) block based mapping.

Since obtaining the results presented in this paper we have continued optimizing our code in an effort to further reduce the size of the state space and the memory usage. Depending on the model variant, we have been able to achieve reductions in memory usage up to 50%.

Our test harness currently always starts with the same initial state of the file store. This is going to be changed so that the initial state can be read from an input file. Files containing valid initial states will be generated through simulation runs.

The capabilities of model checkers have recently been expanded with multi-threading [19], opening up the possibility of significant performance gains. The continued growth of memory sizes will allow us to test longer sequences, and/or increase the complexity of the model.

References

1. ABZ conference: case study details. `http://www.cs.york.ac.uk/circus/mc/abz/`.
2. ABZ conference, October 2008. `http://www.abz2008.org/`.
3. K. Arkoudas. Athena. `http://www.cag.csail.mit.edu/~kostas/dpls/athena/`.
4. K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. On verifying a file system implementation. In *Formal Methods and Software Engineering*, volume 3308 of *LNCS*, pages 373–390. Springer Verlag, 2004.

5. J. C. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Aspects of Computing*, 18(2):143–151, 2006.
6. M. Butler. Some filestore developments with Event-B and RODIN. Workshop at ICFEM, 2007.
7. A. Butterfield and J. Woodcock. Formalising flash memory: First steps. *ICECCS*, pages 251–260, 2007.
8. Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 212–217, New York, NY, USA, 2007. ACM.
9. E. W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, chapter 1, pages 1–82. Academic Press, 1972.
10. 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*. Newcastle University, CS-TR-1099, May 2008.
11. L. Freitas, Z. Fu, and J. Woodcock. Posix file store in Z/Eves: an experiment in the verified software repository. *ICECCS*, 00:3–14, 2007.
12. L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the verification grand challenge: a roadmap. In *13th Int'l Conference on Engineering Complex Computer Systems (ICECCS 2008)*. IEEE, 2008.
13. A. Galloway, G. Lüttgen, J. T. Mühlberg, and R. I. Siminiceanu. Model-checking the Linux virtual file system. In N. D. Jones and M. Müller-Olm, editors, *Verification, Model Checking and Abstract Interpretation*, volume 5403 of *LNCS*, pages 74–88. Springer Verlag, 2009.
14. Grand Challenge 6. `http://vsr.sourceforge.net/gc6index.htm`.
15. Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE '07: Proceedings of the 29th Int'l conference on Software Engineering*, pages 621–631, Washington, DC, USA, 2007. IEEE Computer Society.
16. T. A. Henzinger et al. Temporal safety-proofs for systems code. In *CAV 2002*, volume 2404 of *LNCS*, pages 526–538. Springer Verlag, 2002.
17. Tony Hoare and Jay Misra. Verified software: theories, tools, experiments, July 2005. `http://vstte.ethz.ch`.
18. G. J. Holzmann. Promela language reference. `http://www.spinroot.com/spin/Man/promela.html`.
19. G. J. Holzmann and D. Bošnački. The design of a multi-core extension of the Spin model checker. *IEEE Transactions on Software Engineering*, 33(10), Oct 2007.
20. Gerard J. Holzmann. An improved reachability analysis technique. *Software Practice and Experience*, 18:137–161, 1988.
21. Gerard J. Holzmann. State compression in SPIN. In *Proc. Third SPIN Workshop*. Twente University, The Netherlands, 1997.
22. Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.
23. Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. New challenges in model checking. In *Proc. Symposium 25 Years of Model Checking*, volume 5000 of *LNCS*, pages 65–76. Springer Verlag, 2006.
24. Gerard J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 2(3):270–278, Nov. 1999.

25. I. Houston and S. King. CICS project report: Experiences and results from the use of Z. In *Proceedings of VDM'91*, volume 551 of *LNCS*. Springer Verlag, 1991.
26. ICFEM Flash File System Workshop. *Modelling Flash Memory*, November 2007.
27. Intel Corporation. *Intel Flash File System Core Reference Guide*, version 1 edition, October 2004.
28. D. Jackson. *Software Abstractions*. The MIT-Press, 2006.
29. Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified software: A grand challenge. *IEEE Computer: Software Technologies*, 39(4):93–95, April 2006.
30. Rajeev Joshi and Gerard J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007.
31. E. Kang and D. Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *Abstract State Machines, B and Z*, volume 5238 of *LNCS*, pages 294–308. Springer Verlag, 2008.
32. Zhanzhan Liu, Lihua Yue, Peng Wei, Peiquan Jin, and Xiaoyan Xiang. An adaptive block-set based management for large-scale flash memory. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1621–1625, New York, NY, USA, 2009. ACM.
33. Carroll Morgan and Bernard Sufrin. Specification of the UNIX filing system. *IEEE Trans. Software Eng.*, 10(2):128–142, 1984.
34. Jan Tobias Mühlberg and Gerald Lüttgen. Blasting Linux code. In *FMICS/PDMC*, pages 211–226, 2006.
35. Part 1: Base definitions POSIX. ISO/IEC 9945-1:2003.
36. Part 2: System Interfaces POSIX. ISO/IEC 9945-2:2003.
37. T. C. Ruys. *Towards Effective Model Checking*. PhD thesis, University of Twente, Enschede, March 2001.
38. J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
39. P. Taverne. Raffs: Model checking a robust abstract flash file store. Master's thesis, Delft University of Technology, 2009. `http://repository.tudelft.nl/view/ir/uuid%3A2b4a1434-8169-481d-9824-fe79e9c4874c/`.
40. Verified software repository. `http://vsr.sourceforge.net`.
41. Verified software: Theories, tools, experiments, October 2005. `http://vstte.inf.ethz.ch/`.
42. Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.

TUD-SERG-2009-033
ISSN 1872-5392