RAFFS: Model Checking a Robust Abstract Flash File Store

Master's Thesis

Paul Taverne

RAFFS: Model Checking a Robust Abstract Flash File Store

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Paul Taverne
born in Leiden, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

RAFFS: Model Checking a Robust Abstract Flash File Store

Author: Paul Taverne
Student id: 1018361
Email: p.taverne@student.tudelft.nl

Abstract

This thesis 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. I have designed a simple file store and implemented it in the form of a Promela model. This file store is robust, meaning it can cope with unexpected power loss without getting corrupted. 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 my implementation. A distinguishing feature of my approach is the exhaustive verification of power loss recovery.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Ir. C. Pronk, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. K.G. Langendoen, Faculty EEMCS, TU Delft

Preface

The work described in this thesis was carried out between October 2007 and July 2009 at Delft University of Technology. I have modeled an abstracted file store and its surrounding eco-system. Using model checking technology I have verified that the file store implementation functions exactly as dictated by its specification. Thanks to a configurable bound on the 'depth' of testing, I have even been able to apply exhaustive model checking, something that is normally infeasible for complex models like the one that I have created. My work demonstrates the capabilities and usefulness of model checking. It also shows the enormous complexities involved with verifying a software implementation of a file store. I am confident that my efforts have been a worthwhile contribution to the work done by the formal methods community and to software engineering in general. A paper based on this thesis has been submitted to ICFEM 2009 [26].

Finalizing this thesis took much longer than required, due to a lack of motivation, and a hobby which turned into a successful business. I would like to thank Kees Pronk, my parents, and my friends for pushing me to complete this thesis.

Paul Taverne
Delft, the Netherlands
August 4, 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is the result of the research that I have conducted in completion of the Master program Computer Science at the Delft University of Technology.

1.1 Verification of Software

Software development is a difficult job. Software design involves complex tasks such as the gathering and definition of requirements. Developers also have to anticipate on how the software will be used and how it will interact with other software. A design is usually documented as a collection of text, pictures and diagrams. The problem with these methods of documentation is that the information is often incomplete and ambiguous. It is therefore not surprising that software implementations often differ from their (intended) design.

Humans make mistakes. Writing software is still mostly done by humans. So even in the ideal situation that a design is clearly defined, it is still likely that an implementation will be incorrect. Programmers spend a large portion of their time towards finding and correcting errors in their software that have been discovered through testing. Various tools have been developed that aid the programmer in finding bugs. For example to find code that references a null-pointer or to discover memory leaks. Heuristic tools can look for *bad smells* in the source code. That are pieces of code that have a high probability of containing a error. Testing and usage of this kind of tools will help to improve the quality of a software product. However, it gives no guarantee that the software is completely free of errors and that the implementation adheres to the specified design [9].

To produce software that is guaranteed to be correct, a different paradigm is needed that uses formal methods. Formal methods can be used to describe and explain software, both to the user and the developer. They produce precise documentation, which can be structured and presented at different levels of abstraction.

Formal methods are currently already being used on a limited scale, mainly in safety-critical applications. Using formal methods should eventually become a common practice in mainstream software development. In addition to other quality assurance methods such as testing.

There are two approaches to using formal methods. *Post facto* and *correctness by construction* [30]. The first approach uses technologies like model checking to deter-

mine the correctness of an (abstraction of an) existing implementation. This is often called *verification*. The second approach can be used to construct an implementation based on a formal specification. This technique is sometimes referred to as *refinement*.

1.2 Verification Grand Challenge

Research about software formalization and verification has to be intensified and become more coordinated. Tony Hoare has therefore proposed a Grand Challenge project [19], which long-term vision is to develop methodologies and a set of automated tools that can be used, during the various steps of the software development process, to verify whether a piece of software meets its requirements. The tools should be applicable to a wide range of software and ideally also to itself [30]. This tool-set has also been referred to as a *Verifying Compiler* [5]. The concept of a verifying compiler was first proposed in 1967 by Robert Floyd [13].

One of the steps towards the realization of the vision presented by Hoare is to build a repository of formalized software designs and verified implementations. The main purpose of this repository [47] is to have a variety of input samples that can be used to test and develop the before mentioned tools.

The first case study for the Verification Grand Challenge was Mondex [47], a smartcard that functions as an electronic purse. The Mondex system allows for secure transactions between two cards, or between a card and a bank. One of the key features of this system is that its transaction protocol is failure proof. If a transaction gets aborted for whatever reason, then no money can disappear or be generated out of thin air. The protocol is also resistant against malicious attacks, such as message replays.

1.3 Mini Challenge

At the VSTTE conference [48] in Zürich in 2005, Gerard Holzmann proposed another pilot project. The goal of this so-called *mini-challenge* [31] is to build a verifiable file system that is specifically designed to work with Flash memory as the storage medium. Holzmann argued that this would be a good candidate for inclusion into the repository because (a) it is of sufficient complexity that traditional methods such as testing and code reviews are inadequate for proving its correctness, (b) it is of sufficient simplicity that the problem can be solved within a few years time, and (c) it would have an impact beyond the verification community.

The project suggests using a subset of the POSIX standard [45]. It also defines strict robustness requirements. These robustness requirements include the ability to recover from faults that are specific to Flash memory, such as bad blocks and bit corruption. The file store should also be able to cope with unexpected power loss, without getting corrupted. After a power loss situation has occurred and the system has been restarted, all changes made by unfinished file store operations should either be completely undone, or operations must be continued to completion. So essentially each operation must appear to be an atomic transaction.

The mini challenge has been embraced by the Formal Methods community. It for example has been a case study for the ABZ conference [2, 1] and has also been a topic

at several VSR-net workshops. My supervising professor, Kees Pronk, has held short presentations about my work at two GC6 workshops [17].

1.4 Research Topic

In this thesis I will present the work that I have done with regard to the mini-challenge. This work is experimental in nature. My approach is unique in that it forms a kind of middle road between the two approaches of using formal methods. I 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 [20]. The environment includes a Flash memory and a test harness that interacts with the file store. For simplicity reasons I have assumed that the Flash memory is fully reliable. My focus has been on the other robustness requirement that was defined in the mini-challenge, namely the ability to cope with power failures. RAFFS is capable of fixing inconsistencies that may be present in the file store after a power failure. My model includes the injection of such power failures. A 'proof of correctness' for my file store implementation is obtained by (exhaustive) model checking. The subtopic of robustness in case of power loss has not yet gotten much attention from the verification community. I think this thesis is a relevant and useful contribution towards the mini-challenge and the grand verification challenge in general.

This thesis is organized as follows. Chapter 2 discusses Flash memory. Specific attention is given to its properties and quirks that are relevant for a file store. Chapter 3 will discuss the modeling language Promela and the model checking tool SPIN. Related work is discussed in chapter 4. The goals, requirements, and assumptions that I have set for my research are discussed in chapter 5. The design of my model is discussed in chapter 6. Chapter 7 will discuss how I have applied testing and model checking on my model. In chapter 8 I will present some measurements related to model checking particulars, such as memory usage and running time. Finally, chapter 9 will present conclusions and ideas for future work.

Chapter 2

Flash Memory

In this chapter I will discuss Flash memory. I will give specific attention to those properties that are relevant from the perspective of a file store design.

2.1 Introduction

Flash memory is a non-volatile form of memory, which means that the stored data is retained in the absence of an electrical current. Flash memory has no moving parts, has low power consumption, and can be packaged in a small and light-weight form. It also has a strong resistance to physical shocks and can operate under a wide range of temperature and pressure levels. These properties make it a useful technology for use in portable and embedded systems. Flash memory is also being used in space, for example in satellites and the mars rover [39].

2.2 Technology: NOR vs. NAND

There are two types of Flash memory. The first is based on NOR transistors, the second is based on NAND transistors. NOR Flash has a lower density than NAND Flash. NOR Flash has a high read speed, but is slow at writing and erasing. NAND Flash on the other hand has a bit lower read speed, but is much faster when it comes to writing and erasing. NOR Flash has a random-access interface, making it suitable for directly executing code. It is therefore mostly used for storing boot, operating system, or application code. Data that is often read and rarely written to. NAND Flash has an indirect interface and must be accessed through a command sequence. Due to its higher density, higher write performance, and lower cost per megabyte, NAND Flash is the preferred type for (mass) data storage.

2.3 Hardware Layout

Flash memory chips are arranged into *blocks*, which are further subdivided into *pages*. This is similar to clusters and sectors on a magnetic hard drive. The size of a page and the number of pages per block for a Flash chip depends on the brand and type. Common sizes are 512 or 2048 bytes for a page, and between 16 and 128 pages per block [8, 41]. Each page also has a few bytes of additional space available, in an area

called the *spare data region* [34] or *reservoir* [42]. This area can be used for storing metadata and error correction data.

The Open NAND Flash Interface Working Group (ONFI) has developed a standard for interfacing with NAND flash chips [43]. This standard includes the discovery of device specific parameters such as the page and block size.

2.4 Memory Operations

There are three types of memory operations: read, program, and erase.

2.4.1 Read

An entire page can be read in a single operation. It is also possible to perform a partial read, where only a subsequence of bytes from the page is read. For example just the contents of the spare data region.

2.4.2 Program

The program operation is used for writing data to a Flash page. Programming can, just like the read operation, be performed on an entire page or on part of a page. In an empty (read: erased) page all bits have logical value 1. The program operation can only change bit values from 1 to 0. This limitation gives no problems the first time an erased page is programmed. But once a page has been programmed, it is no longer possible to overwrite the page with arbitrary data.

2.4.3 Erase

The erase operation resets all bits in a page to logical value 1. The unit of erasure is a block. Erasing a block will affect all pages inside that block. It is not possible to erase individual pages.

2.5 Out of Place Updates

Once a page has been programmed it is not possible to simply overwrite it contents due to the limitations of the program operation. Erasing a page before re-programming it is also not possible because the erase operation works on entire blocks instead of individual pages. A solution for these problems is to write new data to a free location instead of overwriting the old data. The page containing the old data can then subsequently be 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. Page states will be discussed in section 2.8, garbage collection will be discussed in section 2.6. Moving data around on the Flash memory requires some modifications to the file store that uses the Flash memory. This will be discussed in section 2.7.

2.6 Garbage Collection

The erase operation can only be performed on blocks. This means that erasing should only be performed when all pages in a block are no longer being used to store valid data. As long as there is enough free space available there is no immediate need to reclaim space that is being consumed by pages with obsolete content. A garbage collection algorithm is responsible for reclaiming the space that is occupied by obsolete content. Such an algorithm can run as a background process or 'on-demand' of the file store.

It can happen that several pages with obsolete content exist, but those pages are all located in blocks which also contain pages with valid data. That means that none of those blocks can be erased directly. The solution for such a situation would be to copy valid data to another location in a different block and mark the original data as obsolete. This can be repeated until a block contains only pages with obsolete content and can thus finally be erased. Copying data means that garbage collection requires some free space in order to be able to perform its task. Free space is likely to be scarce when the garbage collection is active, so some free space must always be reserved specifically for the garbage collector.

2.7 Flash Translation Layer

For compatibility with existing operating systems and file 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) [28]. Using an FTL is not only useful in combination with traditional file systems. It can also be used in a file system that is designed specifically for Flash.

One of the main tasks of the FTL is to perform the out of place updates that were discussed in section 2.5. Unfortunately, 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. Frequently updating inodes is undesirable and would cause a vicious circle of changes. To prevent this from happening, virtual addresses are used in the file store instead of physical addresses. Every chunk of data is assigned a virtual address. The virtual address remains unchanged, regardless of the physical location of the data, and moves along with the data. From the point of view of the file store, data can be overwritten in its current (virtual) location. The virtual address is stored in the *spare data region* of a Flash page along with a two bit version field. A translation table is maintained in RAM with a mapping from virtual addresses to the corresponding physical addresses. This table can always be re-created, for example during mounting, by reading the contents of the spare data region from all Flash pages.

The protocol of updating a piece of data works as follows:

1. Read version field from spare data region of the page that contains the existing data.

2. Increment version with 1, then perform a modulo 4 operation to always maintain a two bit value.

3. Write new data to a free location. Use the same virtual address and the incremented version.

4. Update the mapping table with the new physical location for the virtual address.

5. Mark the page that contains the old data as obsolete.

If something goes wrong after step 3, but before step 5, then the version field can be used to determine which of the two valid copies of the data is the newest. How a page can be marked as obsolete will be discussed in section 2.8.

2.7.1 Mapping Schemes

Different schemes exist for mapping virtual addresses to physical locations. The simplest mapping scheme is based on virtual page numbers (VPN). Every logical address equals a VPN, which is mapped onto a physical block address (PBA) [8]. A PBA consists of a block number and a page index within that block. So every logical location is mapped onto a physical page. An advantage of this fine-grained mapping scheme is that it is straightforward to implement. A disadvantage is that it results in a large mapping table, which can be undesirable because of its RAM usage.

Another mapping scheme for NAND Flash is called NFTL [46]. This scheme has coarse-grained address translation based on virtual block numbers (VBN). Each logical address now equals a VBN and an index within that VBN. The mapping table translates from VBN to physical blocks. The advantage of this mapping scheme is that it has a much smaller mapping table. The size of the table is equal to the number of blocks in the Flash memory, whereas in the VPN scheme the size is equal to the number of pages. A disadvantage of the VBN scheme is that overwriting data has a huge overhead. As usual when overwriting, data must be relocated to a free page. In this case it must be written to a different block, at the same index within the block. Since multiple logical locations share the same VBN, and there is a one-to-one mapping from VBN to physical blocks, all other content in the same physical block must also be relocated. The high frequency of relocations also results in poor space utilization and more garbage collection.

More complex mapping schemes exist that try to improve on the NTFL scheme. An example is [33], where a replacement block can be shared by multiple logical blocks. This scheme gives better performance and space utilization than NFTL.

2.8 Page States

The spare data region of a Flash page contains three bits that indicate the status of the page. These status bits are for example used when marking (the contents of) a page as obsolete. The three status bits are named (in order): *free_inuse*, *valid_obsolete*, and *erased_written*. A bit value of 1 means that the first keyword in the name applies, a value of 0 means the second keyword applies. For example value 010 corresponds to

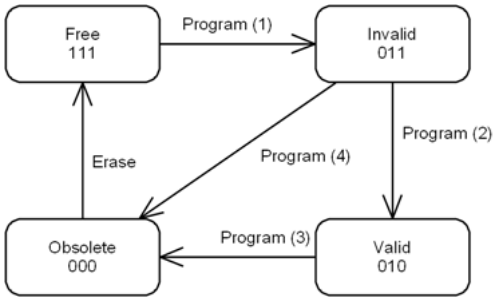

Figure 2.1: Page States

the combination inuse/valid/written. Figure 2.1 shows all page states and the allowed transitions between those states.

As you can see in the diagram, it takes two program operations to transform a page with state *free* into a page with state *valid*. The Flash program operation is not atomic. It consists of a few steps in which each time a few bytes are written atomically. If an empty page is being programmed and a loss of power occurs, then the validity of the contents of the page is unknown. So for robustness reasons, writing data to an empty page is a two step protocol [27]. During the first program operation, the desired content is written, and the page state is set to *invalid*. The status bits in the spare data region are the first bits to be programmed. After the first program operation, the spare data region of the page is programmed a second time. The only modification that is made to the contents of the page is flipping the value of the *erased_written* status bit, changing the page status to *valid*. This minor change is an atomic operation. The contents of the page is now guaranteed to be valid. Marking a page as obsolete is done by programming it a third time. This time the value of the *valid_obsolete* status bit is flipped.

2.9 Reliability

Flash memory is well known to have some reliability problems, such as bit flips and bad blocks. Electrical interference in the memory can cause bits to randomly flip value. An error detection and correction mechanism should be used to combat such data corruption.

The erase and program operations degrade the oxide material of the Flash memory. This deterioration can eventually lead to the memory blocks becoming unreliable and fail. Such a damaged block is called a bad block. The number of erase/program cycles that a block is guaranteed to be able to undergo, is called the *endurance* of the Flash memory [41]. Consumer quality chips have an endurance of 10,000 or more cycles. Industrial quality chips have an endurance of 1,000,000 or more cycles. The deterioration of blocks is called *wear*. To prevent some blocks to go bad much sooner than other blocks, erase/program operations should be distributed as evenly as possible over all blocks. This is called *wear leveling* and is typically done by specialized algorithms.

NAND type Flash memory can have some faulty blocks that are already present during the production process. At the end of the production process the memory is checked for faults and *bad blocks* are marked. All locations on the device initially have all their bits set to a 1, except the invalid blocks [42]. Bad blocks can be found by checking the spare data region of the first two pages in each block. If a zero bit

is found there, it means the block is marked as bad. This information should be used to construct a list of bad blocks. Caution need to be taken with erasing blocks. Bad blocks can sometimes be erasable and accidentally erasing such a block will cause the bad block marking to be lost.

Most manufacturers guarantee that the very first block in the Flash memory is working correctly. This makes it a good spot to store a list of bad blocks, for example in the form of a bitmap. Since it is possible that this 'block 0' becomes bad later, it is recommended to have some kind of backup of the bad block information.

2.10 Flash File Systems

Most commercially available applications of Flash memory use regular file systems such as FAT. Flash specific behavior is then implemented in an FTL such as described in section 2.7. The FTL is typically incorporated in the driver or firmware of the storage device. File systems that have been designed specifically for Flash memory also exist, but are in most cases proprietary solutions. Two dedicated Flash file systems with a publicly available specification are YAFFS (Yet Another Flash File System) [34] and JFFS2 (Journaling Flash File System) [49]. JFFS2 has been designed for NOR type Flash memory. YAFFS is suitable for NAND type Flash memory. I will discuss YAFFS in short below.

2.10.1 YAFFS: Yet Another Flash File System

The main difference between YAFFS and other file systems is the way it stores inodes on the storage medium. In YAFFS, the header of an inode and its list of data locations are stored separately. The header can always be stored in a single page. Instead of letting the inode contain a list of data locations, the data locations contain metadata that point back to the inode. So the inode does not point to the data, but the other way around. The metadata that is used to point back to the inode is a file identifier and a chunk number. In this design there is no need for a mapping table like is used by an FTL. The full inode structure can be rebuild during mounting by making a single pass over the entire memory to read all metadata. YAFFS does not use a mapping table. But other tasks typically done by an FTL, such as out of place updates (section 2.5) and the programming protocol (section 2.8), are also performed by YAFFS.

A downside of the YAFFS method is that all inodes must be kept in RAM to be able to access them quickly. Since the inode data is scattered over the Flash memory, there is no easy way to gather that information from the Flash memory. In the worst case a full pass would need to be taken over the Flash memory to gather all required information to reconstruct a full inode. An optimization is possible to lower the memory requirements at the cost of extra overhead. Instead of using full page addresses in the data location table of an inode, shorter addresses could be used that point to the region where the desired contents can be found. This means that a region of pages must be searched through to find the exact page that contains the desired data.

In file systems where inodes are stored as a whole (possibly spread over multiple pages) it is easy to read an inode from the Flash memory. It is then not needed to constantly keep the entire file tree in RAM. Just the top of the tree is sufficient. The

fact that stored inodes are larger and can span multiple pages means there is some additional overhead when modifying an inode when compared to YAFFS.

Chapter 3

Promela and SPIN

In this chapter I will discuss the modeling language Promela and the model checking tool SPIN, both of which I have used in this thesis project.

3.1 Promela

In the sections below I will give a short introduction to Promela [20]. Section 3.1.1 discusses processes, and section 3.1.2 discusses communication between processes. In section 3.1.3 some differences with the language C are highlighted. Section 3.1.4 discusses the most important language constructs of Promela.

3.1.1 Processes

A Promela model contains one or more processes. Processes are declared globally and can optionally have parameters. There are two ways to start a process. The first method is to include the keyword *active* in the process declaration. This keyword causes the process to be created during initialization of the model. The second method for starting a new process is by using the following statement: *run processname([arguments])*.

The execution of statements from all the running processes are interleaved. During simulation, the interleaving order is either chosen non-deterministically or can be interactively controlled by the user. During verification, all possible interleavings are examined.

A process terminates when it has executed all of its statements. Execution of a process can get suspended, for example due to a blocking statement. A blocking statement is a statement that is not executable given the current state of the model. Such discontinuity of a process is usually temporarily, but can be permanent. For example in a situation were deadlock occurs. A global constraint on the executability of a process can optionally be added to the declaration of a process using the *provided* keyword. When that constraint evaluates to the boolean value true, the process may execute a statement. When the constraint evaluates to false, the execution is suspended. Instead of a global execution constraint, it is also possible to place such execution constraints only at specific places in the code. These constraints are called *guards* (see section 3.1.4). A guard has many uses. For example as a locking mechanism around critical sections. Or as a synchronization mechanism in a distributed algorithm. Guards are also used in selection and repetition constructs, which I will discuss in section 3.1.4.

3.1.2 Communication

Processes can share data through globally declared variables. Communication between processes can be done using channels. A channel is a buffer that stores messages. Processes can send messages to a channel and they can receive messages from a channel. Each message consists of a number of data fields, were each field can have its own data type. The capacity of a channel is specified during its initialization. Messages are by default stored in FIFO order. The sender of a message can however choose to let his message be stored in sorted order.

A process that wants to send a message to a full channel will be suspended until the channel is no longer full. SPIN has a special option to override this behavior. In that case, the send statement will always be executable, but a message will get lost when the channel is full. Attempting to receive from an empty channel will also result in the process getting suspended. There are several syntactic variations of the receive statement. It is possible to receive the first message from a channel, or a random message. The message can either be retained in or be removed from the channel after receiving it. Finally, the receiver can put some constraints on the contents of the message that it wants to receive. For example, by requiring a certain field in the message to have some specific value. Such constraints can of course cause the process to suspend execution if the contents of the channel does not match the constraints.

A channel may get used by any process that can access it within its scope. It is thus not possible for a sender to explicitly specify which process is allowed receive the message. If this functionality is critical then one could choose to create a separate channel for each pair of processes. By using a process identifier as one of the message fields it is possible to send to a specific process on a shared channel. However, the receivers are then responsible for implementing the correct constraints on the messages that they want to receive.

There is a special channel type called a *rendez-vous* channel. This is a channel with a capacity of zero. A message will not be buffered, but it can only be transferred in a synchronous operation.

3.1.3 Promela versus C

Promela is a modeling language with a syntax that has some similarities with the language C. These similarities are mainly in the basic foundations of the language, such as variable declarations, and (boolean) arithmetic. I will not go into details about all the similarities and differences between the two languages. That falls outside the scope of this thesis. I refer interested readers to the Promela Language Reference [20]. However, there are a few differences that I do like to mention:

- Promela does not have the concept of a function. Instead of functions, inline blocks must be used for code structuring. The inline language construct will be discussed in the next section.

- Promela only has two scopes for variables. A global scope to which all processes have access, and a local scope for each process.

- The flow of control in a C program is deterministic (if we ignore functions that use random number generators). Promela has two control constructs, the *IF* and *DO* statements, that can be non-deterministic.

- Expressions that have side effects are allowed in C, but not in Promela.

3.1.4 Special Language Constructs

Below I will give short descriptions of the most important language constructs that are available in Promela.

Unsigned Data Type

Promela has a special data type called *unsigned*. As the name already suggests, this data type can be used to store an unsigned value. What makes this data type so special is that its size is defined by the user in the declaration of the variable. The size of the *unsigned* data type can vary from 1 to 31 bits. The unsigned data type is useful to restrict a variable to a range of values. For example an unsigned variable with a size of 4 bits can only store values between 0 and 15. SPIN will output a warning when a variable under- or overflows during simulation.

Guard

A guard is a statement that evaluates to a boolean value. This value is used to determine if the statement that follows after the guard is executable. When a guard evaluates to false, the next statement can not (yet) be executed. The program counter will stay in front of the guard and the process is suspended. The value of the guard will be re-evaluated each time the process gets a change to (attempt to) execute a statement. Only when the guard evaluates to true, the program counter steps over the guard, allowing execution of the next statement.

Guards play a special role in control flow structures, which will be explained in more detail in the next subsections. Control flow structures can have multiple execution paths, each with their own guard. The execution of the process will be suspended only when all paths are blocked. The program counter will in that case stay at the beginning of the control structure.

IF Control Flow Structure

The *IF* control structure in Promela differs considerably from its C equivalent. The IF statement can have multiple paths, each optionally protected by a guard. One of the executable paths is chosen non-deterministically and execution control will flow through that path. If all paths are blocked then execution will be suspended. The keyword *else* can be used to create an alternative that is always executable. However, this path will only be chosen when all other paths are blocked. An example code segment is shown in figure 3.1.

```
if
:: x > 10 -> // guard
   y = 2;    // statement
:: x > 5  -> // guard
   y = 1;    // statement
:: else   -> // if all other guards are blocking
   y = 0;    // statement
fi;
```

Figure 3.1: Promela example code for IF

DO Control Flow Structure

The *DO* control flow structure is similar to *IF*. The difference is that the flow of control re-enters the *DO* structure instead of exiting the control structure after completing the execution of one of its paths. This looping continues until a *break* statement is executed. An example code segment is shown in figure 3.2.

```
do
:: x < 10 -> // guard
   x++;      // statement
:: x == 5 -> // guard
   x = 10;   // statement
:: else   -> // if all other guards are blocking
   break;    // exit DO construct
do;
```

Figure 3.2: Promela example code for DO

Skip

The statement *skip* is a synonym for the boolean value true. It is a statement that is always executable but makes no changes to the state.

Inline

A Promela *inline* block is very similar to a C preprocessor macro. One difference is that it results in better to read code, since it for example does not need backslashes to merge multiple lines of code. Inline blocks are very useful for structuring and organizing the code of a Promela model, much like functions in other languages. Invoking an inline results in the textual substitution with the body of the inline. The argument names in the inline body are replaced with the names supplied during the invocation. Each invocation leads to the duplication of the code. Another difference between an ordinary macro and an inline block is that line number information is preserved for an inline.

Unless

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. An example code segment is shown in figure 3.3.

```
{
  // main sequence
  statement1;
  statement2;
  statement3;
  statement4;
} unless {
  x == 1;  // escape condition
  // escape sequence
  statement5;
  statement6;
}
```

Figure 3.3: Promela example code for UNLESS

Atomic Sequences

Execution of a Promela model is normally done one statement at a time. There are two language constructs available to group together multiple statements in a single atomic execution step. They are called *atomic* and *d_step* sequences. There are a few differences between the two.

- An *atomic* sequence can contain non-deterministic code. A *d_step* sequence is executed deterministically, meaning that the first non-blocking path is taken in control flow constructs.
- A *d_step* sequence is executed more efficiently than an *atomic* sequence.
- An *atomic* sequence allows goto jumps into and out of it. No goto jumps into or out of a *d_step* sequence are allowed.
- A *d_step* sequence may not contain any statements that can block the execution. An *atomic* sequence may contain blocking statements. However, atomicity is lost when a statement is not executable.
- The behavior within an *unless* statement is different. A *d_step* sequence will always remain atomic within an *unless*, whereas an *atomic* sequence will lose atomicity when the escape condition is triggered.

3.2 SPIN

SPIN [22] is a model checking tool for Promela. It can perform both simulation and verification of a model. During a simulation run, the execution path and process scheduling can either be chosen automatically or can be controlled interactively by the user. For verification purposes, SPIN generates a verifier program in C code. This verifier will attempt to visit the whole state space by examining all possible execution paths within each process and all execution orders of processes. The task of the verifier is to find a counter-example if one exists. There are three types of correctness criteria which can be checked by the verifier. The first type is *safety*. This includes assertions in the code and checking on invalid end states. The second type is *liveness*. This can be used to for example check for infinite loops. The third type is temporal properties. Linear temporal logic (LTL) formula can be used to check temporal behavior.

3.2.1 Verification Modes

SPIN offers three verification modes: *exhaustive*, *bitstate hashing*, and *hash-compact*. Exhaustive will explore the entire state space, provided that enough memory is available. The bitstate and hash-compact modes are approximative, meaning that are they might not visit all reachable states. How each mode works is explained shortly in the subsections below. A more advanced algorithm has recently been devised by Dillinger and Manolios, which can dynamically adapt itself to the size of the state space [11]. This algorithm will hopefully get incorporated in a future version of SPIN as a replacement for or addition to the current approximative modes.

Exhaustive

In exhaustive mode, all reachable states are stored in a hash table. Within each table slot, states are stored in a linked list. Each state is stored in the form of a *state vector*, which contains the values of all global and local variables, plus the program counters of each process. The purpose of the hash table is to be able to quickly check if a certain state has been visited before. States that have been previously visited will not be explored again. If the hash table is chosen too large, then memory is wasted. If the hash table is chosen too small, then time is wasted because on average a longer linked list of states must be searched through.

SPIN offers various lossless compression methods for reducing the memory requirements needed for storing the reached states. For example *Collapse* and *Minimized Automaton* [22].

Bitstate Hashing

The bitstate hashing mode also uses a hash table. The difference is in the way the states are stored. Instead of storing the state vector, only k bits are stored for each state. These k bits are the result of k independent hash functions. Because a state is stored using just a few bits, this verification mode is very memory efficient. A formula for calculating the probability of a hash collision can be found in [22]. This value is directly tied to the state space coverage of the verifier. A formula for calculating the optimal number of hash functions can be found in there as well. This formula depends on the size of the state space, a value which is in most cases initially unknown to the user. A bitstate verification run with a sub-optimal value for k can be needed to acquire that information. Bitstate hashing is the best mode for models with a very large state space. For medium complexity models, it has lower state space coverage than the hash-compact mode.

A limitation of the current bitstate hashing implementation in SPIN is that it can only allocate a hash table that has a size which is a multiple of 2. Since the verifier itself and the underlying OS also need a bit of memory, that limitation means that in the worst case one can only use at most half of the physical memory of the computer for the hash table. Virtual memory has extremely low performance and should therefore not be used. It is desirable to be able to allocate as much of the available memory as possible. That would require a finer granularity. Dillinger and Manolios describe a modification for SPIN that would make any addressable amount of memory possible for use in the bistate hashing algorithm [10].

Hash-compact

Hash-compact simulates a large memory of size 2^n bits, for example $2^{64} \approx 10^{19}$ bits. Each state is hashed into a n-bit value and stored in a regular hash table. This mode is the preferred one for models of medium complexity. It has the ability to reach 100% coverage of the state space under ideal circumstances.

Chapter 4

Related Work

File systems are abundant in computer systems, however the correctness of their implementations is seldom proved. In this chapter I will discuss some work of others that is related to the work done for this thesis. I will first discuss some papers based upon the refinement approach, followed by a few papers based on the verification approach.

In [35] Morgan and Sufrin give a formal specification of the Linux file system using the Z-notation [44]. 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. In [4] Arkoudas et al. 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 [32] use Alloy [29] 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. Other refinement approaches to the challenge are by [12], [6], [14], [7] and [25].

As examples of the verification approach, I mention two important papers. Galloway et al. [16] 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 I have used during the construction of RAFFS. However, due to the well known state explosion problem, they seem unable to perform exhaustive model checking, whereas I 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 [50]. Like in my 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 I have separated out the file store.

Although not applied to file systems, the article by Mühlberg [36] is also an example of exhaustive testing of existing code using model checking.

I have created a POSIX-like file store using a Flash memory. My file system includes files, directories, reads and writes, block erasure and garbage collection. Since I have assumed the Flash memory to be reliable, I did not model wear, bad blocks, and bit corruption.

A distinguishing feature of my model is that it includes the simulation of general system failure in the form of power loss. My file store has been designed to always recover to a consistent state. Were others have mainly focused on the verification of specifications, I have focused on the full verification of an implementation. My implementation supports multiple simultaneous users of the file store.

Chapter 5

Research Topic

In this chapter I will discuss the goals and requirements that I have set for my research. I will also discuss the assumptions that I have made.

5.1 Goals

The goal of my research is to create a verifiable file store for Flash memory. This file store should be able to cope with unexpected power loss without getting corrupted. To reach this goal two steps will be taken. The first step is to design and implement a simple robust file store. The second step is to use model checking technology to verify that the implementation functions as intended.

5.2 Requirements

I have set the following requirements for the file store design, implementation, and model:

- The implementation should be written in the form of a model.
- The model should support both simulation and verification.
- The implementation should encompass the whole eco-system of the file store, including the Flash memory.
- The design should be robust with regard to handling power failures. All aborted operations should either be fully undone or finished during mounting.
- The model should contain a mechanism for injecting failures.
- The design (and implementation) should have a reasonably sized file store API. It should include at least functions for creating and deleting files and directories. It should also contain functions for file manipulation, such as adding and modifying file data. It should contain mount functionality, as this is essential for the robustness requirement. Full POSIX compatibility is the ultimate goal. I have not set this as a requirement because I think it is infeasible to accomplish during this thesis project.

- The design should include a Flash Translation Layer that handles all Flash specific behavior. The reason for this design choice is that a FTL is also commonly used in practice.
- The file store should preferably support multiple simultaneous users.
- The limits in the file store and the size of the Flash memory should be scaled down as far as possible, without sacrificing functionality.
- The above mentioned scaling should be adjustable. For example controlled by constant values.

I have set the following requirements for the verification:

- The correct functioning of the file store API should be verified, both with and without the injection of power failures.
- It should be possible to inject multiple power failures. That way failures during the recovery process can be tested as well.
- Exhaustive model checking should be used whenever possible.

5.3 Assumptions

Below is a list of assumptions that I have made at the beginning of the project.

- The Flash memory is fully reliable. Including bit flips and bad blocks in the model would make it too large and complex. The problem of bit flips and its solution is something that would be suitable for being tested or verified in isolation. Properly testing the management of bad blocks requires a reasonably large sized Flash memory. I intent to create a very small scale model, with a tiny Flash memory.
- Erasing a Flash block is an atomic operation.

5.4 Reasons for using SPIN

One reason for choosing Promela as the implementation language and SPIN as the model checker is that they meet the requirements that I have set in section 5.2. Furthermore, SPIN is a very mature tool that contains many advanced algorithms. Another important reason why I have chosen to use SPIN is because I have prior experience with that software. It was used during one of my master courses. My thesis supervisor also has extensive experience with SPIN. Because of this familiarity with SPIN I have chosen not to explore alternative model checkers. To fully grasp the capabilities and flaws of a model checker one needs to invest considerable time using the tool. I suspect this is one of the primary reasons why the experts in the formal methods and verification community tend to use only their own tools.

Chapter 6

The File Store Model

In this chapter I will discuss the design of my file store and its surrounding environment. Section 6.1 will discuss the design of my model. Section 6.2 outlines the differences between my file store API and POSIX. In section 6.3 several abstractions are discussed. The handling of concurrency between users of the file store is discussed in section 6.4. Constants that control the size of the file store are discussed in section 6.5. Section 6.6 will discuss some implementation optimizations. Finally, section 6.7 gives an overview of the size of the implementation.

6.1 Layered Design

The model that I have constructed has a layered design, consisting of five layers. The UML diagram in figure 6.1 shows these layers. The arrows in the diagram show which layer uses functionality from another layer. The bottom two layers are the Flash memory and its driver, to be discussed in section 6.1.1. The next two layers belong form the file store, named RAFFS, which is discussed in sections 6.1.2 and 6.1.3. The top layer serves as a test harness for RAFFS. This test harness will be discussed in chapter 7. A concise specification for RAFFS and the underlying Flash layers can be found in Appendix B.

6.1.1 Flash Memory and its Driver

The bottom layer in my model is the Flash memory. I have modeled the Flash memory 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. This is needed for the assumption that an inode always fits in a single page, see section 6.3. 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. This layer handles all interactions with the Flash memory. It exposes functions for reading and programming a page, and for erasing a block. An UML diagram for the Flash layers is show in figure 6.2. The data read and programmed by the Flash driver is stored in a dedicated buffer called the page buffer, which is a local variable of the process that uses the file store.

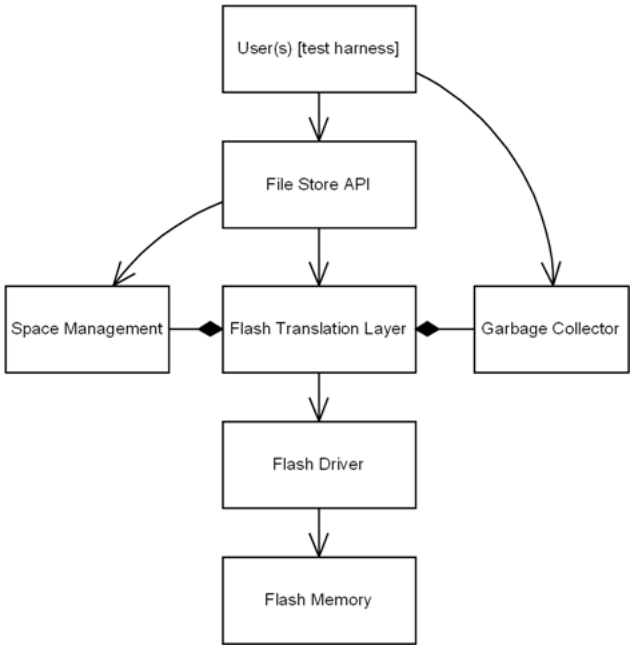

Figure 6.1: UML layer diagram

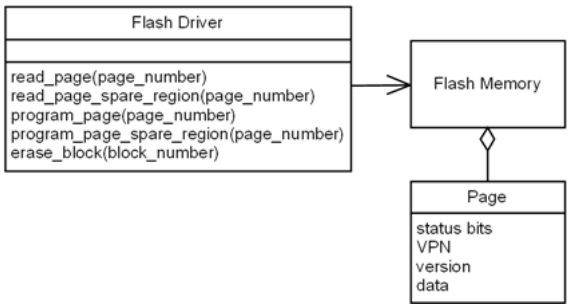

Figure 6.2: Flash memory and driver

6.1.2 Flash Translation Layer

The third layer in the model is the Flash Translation Layer (FTL). The FTL implements all Flash specific behavior and emulates a generic block based storage device. It exposes functions to the file store API layer for writing and erasing data in a virtual location. The FTL maps virtual addresses to physical addresses and performs *out of place updates* as explained in section 2.5. The mapping table is discussed in more detail below. The FTL is also responsible for applying the two step programming protocol when writing data to a Flash page (see section 2.8). The use of that protocol is required for robustness.

The UML diagram in figure 6.3 shows the 'functions' that this layer exposes.

Mapping Table

The mapping scheme that I use in the model treats every logical address as a virtual one and maps it onto a physical page. This virtual address is called a *virtual page number* (VPN). There are three reasons why I have chosen this mapping scheme. First of all, it is the most simple scheme to implement. Secondly, it is compatible with the small

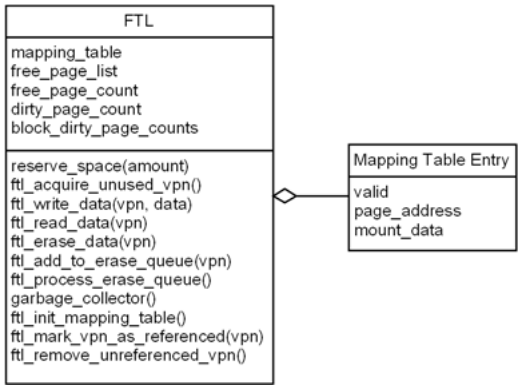

Figure 6.3: Flash Translation Layer

Flash memory that I use. For example, the scheme based on virtual block numbers (see section 2.7.1) would have required additional blocks. That would increase memory usage more than the reduction effect of having a smaller sized mapping table. The third reason is that the VPN scheme requires less copies of page content than other schemes.

Garbage Collector

The FTL contains a simple garbage collection algorithm for recovering space occupied by pages with obsolete content. The algorithm works as follows:

1. Check if the number of free pages is below a threshold value. If there is more free space, then quit.

2. Find the block with the most obsolete pages. The FTL maintains a count of obsolete pages for each dirty block.

3. Read the spare data region of each page in this block. For each page with state *valid*, copy its contents to a free page in another block. Then mark the current page as obsolete.

4. Erase the block.

5. Go back to step 1.

The garbage collection algorithm requires a bit of free space to function properly. In the worst case scenario, only one page is obsolete in a dirty block. In this case, the content of all other pages in that block must be copied to free pages in other blocks. This means that the garbage collection algorithm needs at most 'the number of pages per block minus one' free pages. This number of free pages must always be reserved for use by the garbage collector.

The garbage collector ideally runs on demand of the FTL whenever free space is low. Running it as a separate process is undesirable because adding a process significantly increases the complexity of the model. For optimization reasons I have moved the calls to the garbage collection out of the FTL into the test harness. This is discussed in more detail in section 6.6.

Space Management

The FTL in my design is responsible for the management of free space. It maintains a list of free pages on the Flash memory. It also keeps track of which blocks contain obsolete pages. A block that contains obsolete content is called *dirty* and may at some point in time get processed by the garbage collector. For each block the count of obsolete pages is maintained. This allows the garbage collector to easily find the most dirty blocks.

The mapping table keeps track of which VPN values are in use and which are not. Letting the file store API also keep track of which virtual locations are free would be redundant. I have therefore chosen to only let the FTL manage free space. This also makes it easier to reserve space for the garbage collector. To ensure that enough free space is available to successfully complete a file store operation, each API function must reserve the number of pages that it is planning to use. This number is equal to the number of writes that the function will make.

6.1.3 RAFFS API

The fourth layer in the model is the file store API. The functions that are present in this API are listed in figure 6.4 and are discussed in section 6.2. The functions in the API have a generic implementation and they do not contain any code that is specific for the Flash storage medium.

The mount algorithm and the order in which other API functions write data are of vital importance for the robustness of my file store. This is explained in detail below.

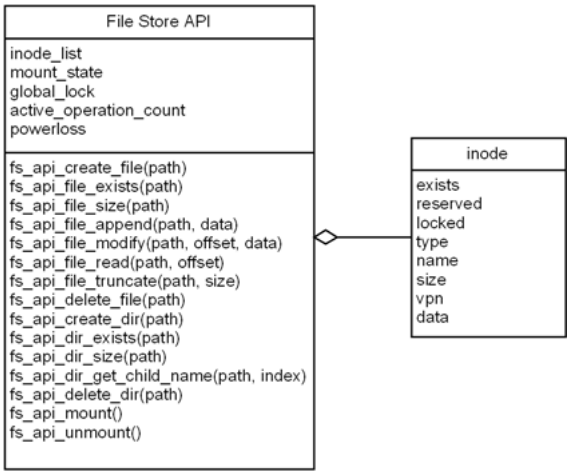

Figure 6.4: File store API

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. It does this with help of the FTL.

My 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, I 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*.

I 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. When an inode is not referenced by another inode (and is not the root), then it is called an orphan. Unreferenced data pages are orphans as well. 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 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 my 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.

Mount Algorithm

The mount algorithm works as follows:

1. Let the FTL initialize its mapping table. This involves reading the spare data region of every page in the Flash memory. Pages with state *free* are added to the free page list (a bitvector). The free space count is also incremented. Pages with state *invalid* are marked as obsolete and the dirty block information is updated. For pages with state *invalid* the dirty block information is updated. For pages with state *valid* an entry in the mapping table is made with the physical address of the page. The index in the table is equal to the VPN value in the page. The value of the version field is remembered in case the same VPN value is encountered a second time. When that happens the page with the oldest version is

marked as obsolete. The address of the page with the newest version is stored in the mapping table.

2. Read the inode of the root directory and store it in the inode list that is kept in RAM. The root inode is always located at VPN 0. A flag is set in the inode to indicate that it needs further processing (see step 3). The FTL is notified that the VPN value of this inode has been referenced (see step 5).

3. For each directory inode in the inode list do the following if it was flagged for processing: Read all child inodes. Flag them for further processing. Store the inode identifier values in the parent inode. Notify the FTL that their VPN values have been referenced. For each file inode in the inode list do the following if it was flagged for processing: Notify the FTL that the VPN values of all the file's data page have been referenced.

4. Repeat step 3 until all inodes have been processed.

5. Let the FTL mark all pages belonging to valid, but unreferenced VPN values as obsolete.

6. Set the system state to 'mounted'.

6.2 POSIX Compatibility

The mini-challenge project suggests using a subset of the POSIX standard [37, 38, 15]. My abstracted file store API is not fully POSIX compatible. I will compare my API with the abstract formal specification [35] from Morgan and Sufrin and discuss which simplications I have made. An overview of the file store API functions in RAFFS is given in figure 6.5.

```
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();
```

Figure 6.5: File store API function prototypes

There are some differences between my 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 my 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. It also allows better downscaling of the file store because

storing a directory inode will require only a single page of Flash memory instead of multiple.

My model does not contain *file descriptors*, or *channels* as they are called in the formal specification. I have abstracted those away to reduce memory usage. Thus the functions *open*, *close*, and *seek* are not present in my model. Instead of a file descriptor, all functions in my API have a parameter *path* which is used to uniquely identify the inode on which the function operates. My model contains no separate naming system like in the formal specification. Each inode always has exactly one name associated with it. As a consequence, my model lacks the *link* and *unlink* functions.

The handling of data also differs from the formal specification. The API functions in my file store 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. Supporting sequences of data as input and output is a possible future extension of the current model. The steps taken to ensure robustness of the file store, as discussed in section 6.1.3, are compatible with such an extension.

The formal specification lacks clear definitions of error conditions. I have defined a set of error conditions for each API function. The implementation checks for those conditions and returns appropriate error codes.

6.3 Abstractions

One of the goals of this thesis project is to perform (exhaustive) verification of my 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 I could accomplish in a short period of time. Others have abstracted parts of an existing file store [16] which proved to be a difficult task. I have decided to design a basic file store from scratch. So instead of abstracting a complex design, I have directly made an abstracted design. The advantage of this method is that I had complete freedom in the design choices, allowing to keep things simple, while also making a robust design. I was not forced to think within the paradigm of other designers. I am confident that this choice has resulted in a model with a much lower complexity than could have been achieved when I 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 my work with other designs and implementations.

There are many abstractions in my 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 I 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. I 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 and limits in the model are specified by constant values. This allows me to easily modify those parameters when desired. Some important constants in the model are discussed in section 6.5.

6.4 Concurrency

A user in my model is a Promela process which contains calls to file store API functions. Multiple concurrent users are allowed to be present in my model. I 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. This is done to prevent common concurrency problems such as *lost updates*. All the file store metadata is shared between the users and is stored as global variables. Locking is for example applied to inodes.

Every Promela process is a big blob of code. The processes include the same inline blocks, which means code is duplicated. Local variables are used for all data that is not meant to be shared between the users. This includes variables that are used in the (duplicated) code for storing intermediate values, such a loop indexes. Another example of a variable that is local to each user process is the one that stores the return value of the file store API functions.

I have chosen to let each user perform all operations itself to maximize concurrency. It is now for example possible to have multiple unfinished operations when a power loss situation occurs. This design also required just minor changes compared to a pure single user implementation.

6.5 Constants

The scale of the file store and the Flash memory are controlled by a few constants. Choosing proper values for these constants is of vital importance. They should be chosen as small as possible to keep the complexity of the model low. But they should also be sufficiently large to maintain full functionality of the file store. For example, all error conditions in the file store API functions must be reachable.

I have listed the most important constants and their current values in table 6.1. A maximum of 4 inodes (including the root) is sufficiently large to be able to create several different directory trees. Only two bits are then needed to uniquely identify an inode. Multiple Flash blocks and multiple pages per block are required to properly test the garbage collection algorithm. A total of eight pages (4x2) was the most optimal choice. That is large enough to be able to store 4 inodes and have room to spare, but also small enough to be able to completely fill it with inodes and data. A page address will fit in just 3 bits. The maximum size of a file is 3 data units. The maximum size of

Name	Value
Maximum number of inodes	4
Maximum file size	3
Maximum directory size	2
Number of data units per page	2
Data unit size (in bits)	1
Number of blocks in Flash memory	4
Pages per block	2

Table 6.1: Important constants

a directory is 2. Using a maximum directory size of 3 would be wrong in combination with the other chosen values, because then at least one error condition would possibly be unreachable. Attempting to add a fourth child to a directory could then, depending on the order of error conditions, trigger an error regarding the maximum number of inodes, leaving the error condition on the directory size untested. I use a test suite (see section 7.1) to confirm the reachability of all error conditions.

6.6 Implementation Optimizations

Several abstractions and optimizations have been previously discussed. Below I will give details about some additional optimizations that I have applied to the implementation of my model.

6.6.1 Garbage Collection

I initially let the FTL call the garbage collector when the amount of free space was low. Adding a separate process for garbage collection is not a viable option because that would significantly increase the complexity of the model during verification. However, since the code from the FTL is inline copied into the file store API functions, which in turn may also have several copies in the user layer, there were several duplicates of the garbage collection code. This had a very negative effect on the compilation time of the verifier. I have therefore chosen to let the user layer explicitly call the garbage collector instead of performing garbage collection automatically in the background. This results in fewer copies of the code and much faster compilation. I have made sure that these changes have no undesired effect on the functioning of the file store.

6.6.2 Atomic Blocks

Another optimization involves the use of atomic blocks. Memory usage is partly determined by the number of states stored during verification. Reducing the number of stored states can reduce memory usage. I have made aggressive use of the Promela *d_step* and *atomic* constructs to group statements together into atomic blocks. I have been able to reduce memory usage significantly with this optimization strategy.

There are two general situations in which *d_step* can not be used, and *atomic* must be used instead. The first situation is non-deterministic code, which would become deterministic if placed within a *d_step*. The second situation is code with blocking statements. In my model blocking statements are for example used for locking.

Every single modification to the contents of the Flash memory in my model must be done in a separate step. This is required for a proper verification of the robustness quality of my implementation in case of power failures. Reading from Flash does not make any modifications to the Flash. Multiple reads can be safely grouped together, since locking mechanisms prevent other processes from accessing those same data locations.

6.6.3 Printf Statements

The use of *printf* statements is useful for debugging purposes to print additional information to the output log during simulation. However, since it is a non-essential statement, it can be worthwhile to remove printf statements when creating a verifier. Tests done on my model showed that removing printf statements can result in a significant reduction of both the size of the state space and the execution time. I have seen reductions in memory use up to 20%. That is a lot considering that I haven't made excessive use of printf. The presence of printf statements in my code has been made configurable through the use of some preprocessor trickery. The relevant code is show in figure 6.6.

```
/* Configuration option for the use of printf. */
#define USE_PRINTF 1

/* Note: no semicolon should be placed after PRINTF_O() in the code! */
#if USE_PRINTF
  #define PRINTF_O(...) printf(__VA_ARGS__);
#else
  #define PRINTF_O(...)
#endif

inline example_code()
{
        PRINTF_O("Optionally print this text.")
}
```

Figure 6.6: Code for optional printf

6.6.4 Hidden Variables

I use a few hidden variables in my implementation. A hidden variable is a variable that is not stored in the state vector. Its value only has meaning within a state. Any assigned value gets lost when a transition to another state is made. Hidden variables are useful for storing intermediate values in an atomic block. There is one limitation with hidden variables, they can not be used in a model if one wants to generate a multi-threaded verifier. I have therefore made the use of the keyword *hidden* optional, so that my model can be made compatible for verification with multiple CPU cores [21].

6.6.5 Locking

Locking mechanisms are used in my model to ensure proper concurrency when using multiple user processes. However, using locking in the single user variants of my model is unnecessary, and would only cause an increase in complexity. I have therefore

disabled all locking code by default in the single user variants of my model. For testing purposes I have added an option to force the inclusion of the locking code.

6.7 Implementation Size

The numbers below give an indication of the size of my model. The values represent the lines of code (LOC) and include white space and comments.

- Variable declaration (including CCVS): 725
- RAFFS plus underlying Flash layers: 2500
- Constants and model options: 430
- All variants of the test harness (including the reference implementation): 3150
- Reference implementation (part of the test harness): 850
- Test suite: 1300

The total size of the code is approximately 8100 LOC and has been divided over 26 source files. I have used Subversion as revision control software for both my model and all relevant documents such as this thesis.

Chapter 7

Testing and Verification Method

The main task of a simple file store is to store a set of files and directories. How it stores that set is hidden from the user(s) of the file store. A user interacts with the file store through an API. This API contains functions for getting information about the items in the set of files and directories, such as getting the size of a file or reading its contents. There are also functions for manipulating the set, such as adding or removing files. If the file store was implemented correctly then the API functions should always behave exactly as described in the file store specification. This conformance to the specification is what I have verified for my implementation using model checking technologies. In this chapter I will discuss how I have applied testing and model checking to my file store implementation.

7.1 Test Suite

It is a good development practice to perform unit testing. Each function and layer in the software should be tested individually, instead of just the system as a whole. This ensures more thorough testing and allows testing of parts before the whole has been completed. Unit testing also helps to approximate the location of a discovered failure, since its cause is likely to be located in a specific portion of the source code. Furthermore, unit testing allows to quickly find regressions when modifying previously tested code.

Applying unit testing to my model is not straightforward. The first problem is that the Promela language lacks functions. Each Promela process is essentially just one big blob of code. Fortunately, by using inline code blocks, it is possible apply structure to the (unpreprocessed) code. This allows re-use of these blocks of code. For example to use them in a dedicated test process. A second and more severe problem is that most of the code in my model has a high data dependency. This means that the behavior of pieces of code not only depend on some input parameters, but also on the state of several (global) variables. Initializing all those variables with proper values for each test case is a very time consuming task. I have therefore chosen to apply unit testing to a subset of the functional units in my implementation. This subset includes all functions from the Flash driver layer, the garbage collection algorithm from the FTL layer, and all API functions from the file store layer. The only functional units that have not been individually tested are located in the FTL layer. These are however

indirectly tested by the file store API unit tests. Furthermore, I have applied model checking on the whole implementation.

I have grouped together all tests in a test suite. This is a variant of my model in which a single Promela process performs a series of tests. The tests of the different functional units have all been put in separate inline blocks. This makes it easy to change which tests are included in the test suite.

The tests for the file store API layer all consist of one ore more API calls. Assertions are used to check if the result of each performed API call matches the expected result. Before each test of the API layer, all variables in the model are (re)set to a state that represents a mounted system. In this initial state, only an empty root directory exists. The remainder of the Flash memory consists of empty pages.

A requirement that I have chosen for the unit tests of the file store API layer is to trigger every error condition in each API function at least once. This is not just important from a testing perspective, but also from a point of view of the capabilities of the file store in the model. A few constants in the model (see section 6.5) control the scale of the file store and the Flash memory. Choosing a wrong value for one of these constants can make it impossible for certain error conditions to be triggered. For example, attempting to surpass the maximum size of a directory could be impossible when the maximum allowed number of inodes in the file store was chosen too small, or when the maximum directory size was chosen too large.

I have used the described deterministic test suite during the early development stages. Once the implementation was nearing completion I additionally starting using a test harness (section 7.4) with non-determinism.

7.2 Code Coverage

SPIN has built-in functionality for measuring code coverage. This functionality is very basic and supplies only the most essential information. It does for example not show the frequency of use for each line of code. It also does not show which percentage of the total code was covered. The only information that is gives are the line numbers and corresponding code segments that were unreachable during a verification run of the model. This basic information is useful to have, since there are no plugins or alternative tools for measuring code coverage of a Promela model. There are three problems with the functionality provided by SPIN. The first problem occurs when code is spread over multiple source files. The line numbers that SPIN spews out for uncovered lines of code are lacking the name of the source file. This makes it cumbersome to find the matching line in your source code. Displaying the file name would be a trivial but useful fix for a future version of SPIN. The second problem is more severe. SPIN gives false positives. This is particularly obvious for pieces of code with a single control flow path, in which a single line is marked as unreachable. I have confirmed my suspicion by deliberately inserting an assertion at a point where a false positive was given. This assertion was triggered as expected during verification, yet the corresponding code was marked as unreachable by SPIN. Due to the untrustworthy coverage data that SPIN provides, I have made little use of this functionality. The false positives seem to be caused by a case caching optimization that SPIN performs by default. A new version (5.2.0) has recently been released that allows disabling that particular optimization.

The third problem occurs with code duplication. Coverage is determined for each duplicate of the code. While this may be desirable in some cases, it can be annoying in others. For example, in my deterministic test suite SPIN will report lots of unreachable code. In this situation I would prefer if SPIN could merge the coverage data of the duplicates. An option to control this coverage behavior would be useful.

7.3 Assertions

I have not only used assertions in the test suite, but also within the implementation of the file store itself. There are 5 situations in which I have used assertions.

At several places in the code assumptions are made that a certain condition is always true. At such places an assertion is placed in the code to verify that the assumption is always correct. For example, when reading a page from the Flash memory that is supposed to have state *valid*, this state value is checked using an assertion.

There are also situations that should never occur. For example, if an array is searched for a specific value that must be located somewhere in the array, then it should never happen that this value is not found. I trigger an assertion failure when such an unexpected situation occurs.

There are many loops in the model. For reasons of memory efficiency, I only use a few different variables as indexes for all loops in the entire model, all in the same (local) variable scope. Remember that Promela is limited to just two variable scopes. The same loop variables are used at several different places in the model. Since nested loops do occur in the model, there is a possibility that a loop variable is accidentally used in both an inner and outer loop. That would of course be a bug. Since I make heavy use of inline blocks for code structuring, it is not unlikely that such a bug would be present in the code. I have taken two measures to detect such bugs. Before each loop I use an assertion to check whether the loop variable has value zero. After each loop I reset the value of the loop variable to zero. Accidental use of a loop variable that contains useful data will trigger an assertion failure during model checking thanks to these measures.

In the test harness, see section 7.4, a series of consistency checks is performed in between each file store API call. These checks mainly involve comparing the metadata that is kept in memory by the file store to the metadata that is stored on the Flash memory. An example is the translation table that contains a mapping from Virtual Page Numbers to physical page addresses. That information should match the metadata that is stored in each Flash page. Assertions are used for all consistency checks.

Only specific transitions between page states are allowed in the model. For example, a page with state *valid* may only become *obsolete*. Assertions in the code belonging to the Flash program operation check for the validity of the page state transitions. I do this by analyzing the current page state and the new page state value before performing the assignment in which the new state value is set. This temporal property could also have been checked using Linear Temporal Logic (LTL). However, that would be extremely inefficient because LTL adds a process to the model. Such an extra process would cause the state space to explode, whereas the assertions can be added at (almost) zero cost inside an already existing atomic step.

7.4 Test Harness

The fifth and topmost layer in my model is the user layer, which functions as a test harness for my file store implementation. I have made several variations of my model, each with a different testing purpose, and with a different implementation of the user layer. Four variants will be discussed in detail; *SU* (single user), *SUPL* (single user with power loss), *MU* (multiple users), and *MUPL* (multiple users with power loss). The first variant is discussed in section 7.5, the second in section 7.6, and the last two in section 7.7.

7.5 SU Variant

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 my model contains a single user process that performs a random sequence of file store API calls. The contents of the Flash memory is initially set to reflect a formatted system, meaning it contains only an empty root directory as stored data. The file store data in RAM is set to match the contents of the Flash memory. In each step of the sequence, a file store API function and its inputs are chosen non-deterministically. Performing full verification on this model will, given a long enough sequence, examine the entire state space.

The behavior of every file store API call should be exactly as described in the specification. I verify this by comparing the return value of each API call with the return value given by a reference implementation. 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 my 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 used. Holzmann used an external reference implementation during randomized differential testing of his own Flash file store model [18].

Because the reference implementation needs to be integrated into the model and because my file store API is unique, I was forced to make my own reference implementation (see section 7.5.2). It has thus also been written in Promela. My reference implementation has not been verified to be correct, I 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 would be unlikely. The only similarities in RAFFS compared to the reference are found in the file store API layer. Obvious bugs in that layer will be found by the test suite.

A bug in either implementation will cause one or multiple of the following situations to occur:

- A file store API function returns an unexpected value.
- An internal assertion failure occurs.

- The set of files/directories as stored in the system no longer matches the set that the reference implementation maintains.

- There is an inconsistency between the data that the file system maintains internally and the data that is present on the Flash memory.

All four situations will (eventually) trigger an assertion failure.

Besides comparing results with a reference implementation, I 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. Inodes and the mapping table of the FTL are examples of such metadata.

The sequences of API calls have a configurable length. This allows me to control the 'depth' of the verification. Increasing the sequence length will increase the size of the state space and thus the complexity of the model. This bound is discussed in more detail in section 7.5.1.

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. Such 'invalid' inputs are tested in the deterministic test suite that was discussed in section 7.1. The obvious reason for restricting the domain of input values is to minimize state space explosion.

```
PROCESS_USER() {
  initialize_system;
  mount_system;
  while(!sequence_complete) {
    generate_inputs;
    choose_api_function;
    perform_api_call_reference;
    perform_api_call;
    compare_results;
    perform_consistency_checks;
    garbage_collection;
  }
}
```

Figure 7.1: Pseudo code for SU test harness

Figure 7.1 shows pseudo code for the model variant SU. I will now summarize in detail how the test operates. The test begins with initializing a known state. This starting state is a formatted system. That means that only an empty root directory is present. Once this initial state has been initialized for both the file store and the reference implementation, a sequence of file store API calls will be performed. This is done in a loop. During each iteration of the loop, the following tasks are performed:

1. Non-deterministically generate values for the input parameters. Not all parameters may actually get used depending on the API call that will be performed. Only generating values for the required inputs requires additional code. I tested if generating only the required inputs resulted in a smaller state space, and it turned out that the state space did not decrease in size.

2. Non-deterministically select a file store API function.

3. Let the reference implementation perform the selected API function and determine the expected return value. The reference implementation will update the set of files/directories that it maintains, as well as the amount of expected remaining free space. The maintained data now represents the state that the system is expected to have after the API call has completed. If the API function is being expected to return an error code, then the expected state will remain unchanged.

4. Let the normal implementation perform the selected API call with the generated inputs.

5. Compare the return values of the two implementations inside an assertion.

6. Perform consistency checks. Metadata that is maintained in RAM by the file store is compared to the metadata stored in the Flash memory.

7. Perform garbage collection. The reason why the garbage collector is called from the user layer was explained in section 6.6. Garbage collection is performed after each API call to ensure sufficient free pages are available. Please note that the reference implementation does not use (or need) garbage collection.

My method has some similarities with work done by Holzmann et al. at Nasa's Jet Propulsion Laboratory [18]. Holzmann also uses a reference implementation and compares its output with that of his own implementation. However, he used a third party reference implementation, not a self-made abstracted reference implementation like I have used. The *randomized differential testing* approach used by Holzmann focuses on smart random testing of a complex implementation, while I attempt to perform exhaustive testing on a much simpler implementation.

7.5.1 Bounded Model

A common issue that one may encounter during model checking is that a model has a very large state space. When the state space is large, performing exhaustive verification is usually impossible. This issue should be taken into consideration when constructing a model, so that measures can be taken to keep both the size of the state vector and the state space as small as possible. I have deliberately kept the scale of the modeled file store small. I have additionally used private variable encoding (see section 8.1) to further reduce the size of the state vector.

Despite my efforts of keeping the complexity of the model low, it is still far to complex to perform exhaustive verification and explore the entire state space. This is what I expected beforehand. I have therefore made the length of the sequence of API calls configurable. By limiting the length of the sequence, the complexity of the model can be reduced significantly. At short lengths, performing exhaustive verification should be possible with minimal memory and runtime requirements.

By simply varying the sequence length, small variations of the model with different complexities can be constructed. This also gives the opportunity to apply the non-exhaustive verification methods that SPIN has to offer. When it is only possible to

cover part of the state space, the challenge is to make that coverage as large as possible. Holzmann suggests using *parallelism* and *search diversity* to increase the problem coverage [24]. This basically comes down to performing multiple verification runs, each with it own specific settings, and divide this work over a number of CPU cores. Holzmann has developed a tool called Swarm [23] that can assist a user in automating these steps.

7.5.2 Reference Implementation

The reference implementation in my 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 regarding those inodes is stored in that list. The inode identifiers are equal to the index in the list. The file store API functions in the reference implementation operate directly on the inode list. The reference implementation is much less complex compared to my other implementation, as there is no storage medium, and no Flash specific behavior. The code size of the reference implementation is approximately one fourth of the size of my RAFFS implementation.

The static information that the reference implementation uses is:

- The specification of the file store. This includes information such as the maximum amount of inodes in the system, the maximum size of a file, and the maximum depth of a path. It also specifies the behavior and error conditions of the file store API functions.

- The amount of usable space that is available on the storage medium. This equals the total space on the Flash memory minus the space that is reserved for the garbage collector.

The dynamic information that the reference implementation uses is:

- The set of files and directories that the file store contains. For each inode, the following information is stored in the list: path, inode type, and size. For file inodes, the data contents of the file is also stored in the list. Directory contents are not stored in the inode, since that information can be deduced from the path values available in the list.

- The amount of free space that is available for use.

The above dynamic information represents the state of the file store. This information is manipulated by the file store API functions of the reference implementation.

7.6 SUPL Variant

A primary goal in my 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 any unfinished operation must

either be completed, or that all changes made by such an unfinished operation must be undone.

A second variant of my model, named *SUPL*, is an extension of the SU variant discussed in section 7.5. The main difference is the addition of power loss in the model. The code that I want to be susceptible to power loss is put inside an *unless* construct (see section 3.1.4). 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 the sequence of API calls is continued. Multiple subsequent power loss situations can occur in the model. They can occur at any time, even during mounting.

```
bool powerloss;
PROCESS_USER() {
  initialize_system;
  while(!sequence_complete) {
    {
      mount;
      select_expected_state;
      perform_consistency_checks;
      while(!sequence_complete) {
        garbage_collection;
        generate_inputs;
        choose_api_function;
        perform_api_call_reference;
        perform_api_call;
        compare_results;
        perform_consistency_checks;
      }
    } unless {
      powerloss == 1
    }
    reboot;
  }
}
PROCESS_TRIGGER_POWERLOSS() {
  powerloss = 1;
}
```

Figure 7.2: Pseudo code for SUPL test harness

Figure 7.2 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. I therefore let the reference implementation remember 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 I 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 of 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.

7.7 MU and MUPL Variants

The verification method that I have described in the previous sections was performed with a single user process. Even though my 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 depends on the execution order of the user processes. The results are therefore not comparable to a reference implementation. I have made two variants of my 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. My main focus with these variants has been on verifying the absence of deadlock. Having multiple processes causes the state space to explode, making it currently impossible to perform exhaustive model checking on the MUPL variant, even with a bounded model. With the MU variant, exhaustive verification is currently only possible with a sequence length of 1. That is far too short to be useful.

Chapter 8

Model Checking Results

The model checking tool SPIN [22] was used for verification of my model. Results of my verification efforts are presented and discussed in sections 8.2 to 8.5. First I will present a method that I have developed for storing variable data in more compact form.

8.1 CCVS: Custom Compact Variable Storage

My 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 [40] I have devised and implemented a generic method for storing the values of multiple variables into a single integer type variable, which then serves as a container. Bit arithmetic is used for reading and writing the values of individual variables from such a container variable. My method is called Custom Compact Variable Storage (CCVS), and makes heavy use of preprocessor macros. The work of Ruys mainly focuses on storing homogeneous variables, such as an array of booleans. I have improved on that by making the implementation extremely flexible, allowing to store a heterogeneous collection of variables, including complex structures.

The code that I have written for CCVS is generic and is therefore suitable for being re-used in other Promela models. The full code plus a usage example can be found in appendix A. I have currently defined variables both as normal individual variables and using my CCVS method. One of these two storage methods is chosen with a preprocessor flag. Being able to turn off CCVS is useful for debugging purposes, since CCVS obfuscates the variables.

The purpose of CCVS is to reduce the size of the state vector and the memory requirements during verification. The effect on the size of the state vector for the different variants of my model can be found in table 8.1. The results presented in the sections below will show the effectiveness of CCVS with regard to memory usage.

8.2 Results for SU Variant

I will first discuss results obtained for the *SU* variant that was described in section 7.4. I have been able to perform exhaustive verification for sequences (of file store API

Variant	Without CCVS	With CCVS	Reduction
SU	140	76	46%
SUPL	160	84	48%
MU	136	84	38%
MUPL	140	88	37%

Table 8.1: State vector sizes (in bytes)

calls) up to length 6 with a memory usage less than 1 GB. Using a different machine, one equipped with 32 GB RAM, I have managed to perform exhaustive verification for lengths up to 8. Sequences of length 15 and 31 have been extensively tested using the bitstate hashing technique.

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 the experience gained by making my deterministic test suite, I 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 I have obtained give me confidence that my implementation functions as intended and expected.

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 8.2: Results for variant SU. Compression: none.

Some results from my exhaustive verification runs can be found in table 8.2, table 8.3, and table 8.4. 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 8.2, *state vector collapse* was used for table 8.3, and *minimized DFA encoding* was used for table 8.4.

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 8.3: Results for variant SU. Compression: -DCOLLAPSE.

CCVS proved to be very useful for reducing memory usage during verification of my model. To my 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 8.4), 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. I have used this powerful compression technique, in combination with CCVS, during the rest of my verification efforts since memory is the most scarce resource available to me.

An interesting observation on the results found in table 8.4 is that the rate at which the state space grows is slowly decreasing as the length increases. Verification with a length of 9 might therefore be possible with a 64 GB machine.

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 8.4: Results for variant SU. Compression: -DMA.

8.3 Results for SUPL Variant

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 7.6. By analyzing all possible execution interleavings of the two processes in the model, I have been able to verify that my 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 8.5: Results for variant SUPL. Compression: -DMA.

I have been able to perform exhaustive verification for a sequence length of 5 on a machine with 32 GB RAM. Some results for verification runs on the SUPL variant are listed in table 8.3. The applied state compression technique was very efficient with a compression of 95.4% at length 5. 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 my case more than acceptable. Being able to trade time for reduction in memory usage has allowed me to perform exhaustive verification for sequences longer than would otherwise have

been possible.

8.4 Results for MU and MUPL Variants

Variant	Users	Seq. Length	States	Memory (MB)	Time (s)
MU	2	4	47,694,148,000	16,386	210,000.0
MU	3	3	67,058,183,000	16,386	328,000.0
MUPL	2	3	46,108,437,000	16,387	288,000.0
MUPL	3	2	67,876,589,000	16,387	428,000.0

Table 8.6: Results for variants MU and MUPL. Bitstate hashing.

I have mainly tested the variants with multiple users (MU/MUPL) for the absence of deadlock in combination with some generic consistency checks. Behavioral correctness of the file store API functions can not be verified as explained in section 7.7. Exhaustive verification turned out to be currently impossible for sequences longer than 1. I have therefore resorted to using bitstate hashing. Results from a few on the performed verification runs can be found in table 8.4.

The reason why only 16 GB of memory has been used in these verification runs is due to a limitation of SPIN which is explained in section 3.2.1. The value of parameter k (the number of bits per state) was set to SPIN's default of 3. I have calculated he optimal value of k using the formula mentioned in section 3.2.1. Given the large state space the optimal value would be 1. I plan to do additional runs with that parameter value. Unfortunately, due to the long execution time of those verification runs those results will not be finished in time for inclusion in this thesis.

8.5 Discussion

I already started using the described verification methods during the development of my implementation. This has helped me to find flaws in my 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 my mistakes, I gained a better understanding of the functioning of the model. This knowledge was beneficial for the remainder of the development cycle, specially with regard to the robustness aspect of my implementation.

I found a few bugs in my implementation through model checking. In my garbage collection algorithm I 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.

Since obtaining the results presented in the above sections, I have managed to optimize my model a bit further. Those optimizations mainly include grouping more parts

of the code together in atomic blocks. This is a powerful method for reducing the complexity of the model, which I had already used frequently throughout the code of my model. It can help to reduce the number of state transitions, the number of stored states, and the number of interleavings between processes. Using basic compression, these additional optimizations give approximately a 20% reduction in memory usage. The benefit might be even higher for the multi-user variants. I don't expect to find much more optimization opportunities. The additional optimizations are most likely not enough to allow verification (with the same hardware) of longer sequences than those presented in the above results. Nevertheless, it makes verification of longer sequences more feasible when internal memory sizes increase in the future. Furthermore, it gives some additional breathing room for reducing abstractions and still being able to exhaustively verify the same lengths using the same hardware.

I have applied *search diversity* to increase the combined state space coverage of multiple verification runs. SPIN offers various randomization options that control how the generated verifier explores the state space. This process can be automated by using a shell script generator called Swarm [23]. It is unfortunately not possible to obtain accurate quantitative results with regard to the state space coverage of a single bitstate verification run, let alone the combined coverage of multiple runs. Bitstate hashing should be considered as a method for attempting to find flaws, not a method to prove the absence of flaws.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

I 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.

A distinguishing feature of my model is 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. A test harness consisting of one or multiple user processes exercises the file store to its full extent by performing non-deterministic sequences of file store API calls.

I have been able to apply exhaustive verification on my model. To achieve this I have reduced the size of the state space by limiting the length of the sequences of file store API calls. One could argue that this restriction implies that the RAFFS implementation itself has not been fully verified, just the bounded model. That is a correct observation. However, due to the small scale of the file store, most interesting situations and corner cases are already reachable with relatively short sequence lengths. I base that conclusion on knowledge gained while working on the model and a deterministic test suite that I have build. Most of the interesting states are reachable with the sequence lengths that I have managed to verify exhaustively. Exhaustive verification of an unbounded model is desirable but still unfeasible with currently available high-end hardware.

This thesis gives quantitative results of the model checking process such as the compression mechanisms used, number of reached states, memory usage, and running time.

The (bounded) exhaustive verification makes my approach stand out from other ways of verifying a file system. In [16] 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 I have used, they seemed unable to use exhaustive verification on their model. Additionally my model includes Flash particulars such as out of place updates and garbage collection. My approach differs from [50] in that I 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 I have

constructed real code, resulting in a functioning system.

My approach of constructing an abstract model in Promela has proven to be very useful to test a design concept on a reduced scale. My model proved to be truly effective in verifying the ability of the RAFFS implementation to recover from power loss.

The RAFFS design and implementation is too abstract and too different from existing file stores to have any direct practical value. This was never the intention. The goal of this thesis project was to investigate and demonstrate the capabilities of model checking in the context of the verification of a file store implementation. Future work could entail moving towards a more realistic design and implementation, or to model an existing real-life implementation.

With current hardware and model checking technology it is extremely difficult, if not impossible, to exhaustively verify a complex software implementation such as a file store. One of the major causes of complexity in a model of a file store is the inevitable dependency on data. Having multiple processes also causes an explosion of the state space.

It has been 4 years since Holzmann's proposed the mini-challenge. His prediction that the problem is solvable within a few years time has turned out to be overly optimistic. A few bits and pieces of the puzzle have been placed in the right positions, but don't foresee the whole puzzle being solved any time soon.

Instead of modeling and verifying a large implementation, an alternative approach would be divide the implementation into several smaller parts and model them in isolation. Proving the correctness of the system as a whole with all parts integrated could then possibly be done with theorem proving on the formal specification of the global design.

9.2 Future Work

I envisage reducing some of the current severe abstractions in the model. A goal is to make my implementation more realistic and more comparable to existing implementations and specifications. I 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 could be replaced by a more complex block based mapping.

My 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. Starting with different initial states can increase the combined state space coverage, without having to change the bound on the model.

Proving the correctness of the reference implementation is also a desirable future task. Having a formally verified reference implementation would further strengthen the confidence in the correctness of my RAFFS implementation. A formal specification of the file store, or at least its API layer, written in a formal language, would be useful asset as well.

Bibliography

[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*, chapter 1, pages 1–82. ACM Classic Book Series, 1972.

[10] Peter C. Dillinger and Panagiotis Manolios. Fast and accurate bitstate verification for SPIN. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software (SPIN)*, pages 57–75. Springer-Verlag, 2004.

[11] Peter C. Dillinger and Panagiotis Manolios. Fast, all-purpose state storage. In *Proceedings of the 16th International SPIN Workshop on Model Checking of Software*, pages 12–31. Springer-Verlag, June 2009.

[12] 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.

[13] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[14] 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.

[15] 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.

[16] A. Galloway, G. Lüttgen, J. T. Mühlberg, and R. I. Siminiceanu. *Model-Checking the Linux Virtual File System*, volume 5403 of *LNCS*, pages 74–88. Springer Verlag, 2009.

[17] Grand Challenge 6. `http://vsr.sourceforge.net/gc6index.htm`

[18] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 621–631, Washington, DC, USA, 2007. IEEE Computer Society.

[19] Tony Hoare and Jay Misra. Verified software: theories, tools, experiments, July 2005.

[20] G. J. Holzmann. Promela language reference. `http://www.spinroot.com/spin/Man/promela.html`

[21] 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.

[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. Swarm verification. In *Automated Software Engineering*, volume ASE 2008, pages 1–6. IEEE, September 2008. `http://spinroot.com/swarm/`

[24] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the swarm tool. In *Model Checking Software: 15th SPIN workshop, UCLA, Los Angeles*, volume 5156 of *LNCS*, pages 134–143. Springer Verlag, August 2008.

[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] International Conference on Formal Engineering Methods, ICFEM '09. `http://icfem09.inf.puc-rio.br/ICFEM.html`

[27] ICFEM Flash File System Workshop. *Modelling Flash Memory*, November 2007.

[28] Intel Corporation. *Intel Flash File System Core Reference Guide*, version 1 edition, October 2004.

[29] D. Jackson. *Software Abstractions*. The MIT-Press, 2006.

[30] Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified software: A grand challenge. *IEEE Computer: Software Technologies*, 39(4):93–95, April 2006.

[31] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007.

[32] E. Kang and D. Jackson. *Formal Modeling and Analysis of a Flash Filesystem in Alloy*, volume 5238 of *LNCS*, pages 294–308. Springer Verlag, 2008.

[33] 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.

[34] Charles Manning. YAFFS specification. `http://www.yaffs.net`

[35] Carroll Morgan and Bernard Sufrin. Specification of the UNIX filing system. *IEEE Trans. Software Eng.*, 10(2):128–142, 1984.

[36] Jan Tobias Mühlberg and Gerald Lüttgen. Blasting Linux code. In *FMICS/PDMC*, pages 211–226, 2006.

[37] Part 1: Base definitions POSIX. ISO/IEC 9945-1:2003.

[38] Part 2: System Interfaces POSIX. ISO/IEC 9945-2:2003.

[39] G. Reeves and T. Neilson. The mars rover spirit flash anomaly. *IEEE Aerospace Conference*, 2005.

[40] T. C. Ruys. *Towards Effective Model Checking*. PhD thesis, University of Twente, Enschede, March 2001.

[41] SAMSUNG Semiconductor. *Application Note for NAND Flash Memory (Revision 2.0)*. `http://www.samsung.com/global/business/semiconductor/products/flash/downloads/applicationnote/app_nand.pdf`

[42] SAMSUNG Semiconductor. *Bad Block Management*.
`http://www.samsung.com/global/business/semiconductor/products/flash/downloads/applicationnote/xsr_v15_badblockmgmt_application_note.pdf`

[43] Hynix Semiconductor. *Open NAND Flash Interface Specification*, 2.1 edition, January 2009.
`http://www.onfi.org`

[44] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[45] Open Group Technical Standard. Protocols for interworking: Xnfs, version 3w. Technical Report C702, The Open Group, February 1998.

[46] Flash-Memory translation layer for NAND flash (NFTL). M-systems, 1998.

[47] Verified software repository.
`http://vsr.sourceforge.net`

[48] Verified software: Theories, tools, experiments, October 2005.
`http://vstte.inf.ethz.ch`

[49] David Woodhouse. JFFS: The journaling flash file system.
`http://sourceware.org/jffs2/`

[50] 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.

Appendix A

Appendix A: CCVS

Below you can find the base code and a usage example for CCVS.

A.1 Code

Figures A.1 to A.3 show the macros that I have used for reading values from a container variable. As you can see both signed and unsigned values are supported. Figures A.4 to A.6 show the macros that I have used for writing values to a container variable.

```
/* Maximum values of some data types that are used in the code. */
#define BYTE_MAX  255
#define SHORT_MAX 32767
#define INT_MAX   2147483647

/* Generic macro for reading a range of bits from an integer data type
 * (byte/short/int). These bits are converted into a value.
 *
 * var: Name of the variable.
 * var_max_size: Maximum value that the data type of the variable can store.
 * startpos: The index of the first bit that must be read.
 * bitlength: The number of bits that must be read.
 *
 * A byte can store 8 bits. Due to signedness, a short can store 15 bits
 * and an int 31 bits.
 */
#define generic_get(var, var_max_size, startpos, bitlength) \
        ((var >> (startpos)) & (~(var_max_size << (bitlength))))
```

Figure A.1: CCVS base code for reading values

```
/* Macros for reading a range of bits from a data type (byte/short/int).
 * These bits are then converted to an unsigned value.
 */
#define byte_get(var, startpos, bitlength) \
        generic_get(var, BYTE_MAX, startpos, bitlength)
#define short_get(var, startpos, bitlength) \
        generic_get(var, SHORT_MAX, startpos, bitlength)
#define int_get(var, startpos, bitlength) \
        generic_get(var, INT_MAX, startpos, bitlength)
```

Figure A.2: CCVS macros for reading unsigned values

```
/* Macros for reading a range of bits from a data type (byte/short/int).
 * These bits are then converted to a signed value.
 */
#define byte_get_signed(var,startpos,bitlength) \
        (generic_get(var,BYTE_MAX,startpos,bitlength) - (1<<(bitlength-1)))
#define short_get_signed(var,startpos,bitlength) \
        (generic_get(var,SHORT_MAX,startpos,bitlength) - (1<<(bitlength-1)))
#define int_get_signed(var,startpos,bitlength) \
        (generic_get(var,INT_MAX,startpos,bitlength) - (1<<(bitlength-1)))
```

Figure A.3: CCVS macros for reading signed values

```
hidden int generic_set_temp_value;

/* Generic macro for storing a (positive) value into a range of bits from an
 * integer data type (byte/short/int).
 *
 * gs_var: Name of the variable in which the value should be stored.
 * gs_var_max_size: Maximum value that the data type of the variable can
 * store.
 * gs_var_bitlength: The number of bits in the variable that can be used for
 * data storage. This is 8 for a byte, 15 for a short and 31 for an int.
 * gs_startpos: The index for the first bit of the range of bits to which the
 * value must be stored.
 * gs_bitlength: The length of the range of bits that will used to store the
 * value.
 * gs_value: The value that must be stored. It must be small enough to fit in
 * the specified range of bits.
 */
inline generic_set(gs_var,gs_var_max_size,gs_var_bitlength,gs_startpos, \
                                          gs_bitlength,gs_value) {
        d_step {
        assert((gs_var_bitlength) >= ((gs_startpos) + (gs_bitlength)));
        /* first evaluate gs_value, otherwise things can go wrong! */
        generic_set_temp_value = gs_value;
        assert(((generic_set_temp_value) >= 0) && \
                ((generic_set_temp_value) < (1 << (gs_bitlength))));
        gs_var = gs_var & (~((gs_var_max_size >> \
                (gs_var_bitlength - gs_bitlength)) << (gs_startpos)));
        gs_var = gs_var | (generic_set_temp_value << (gs_startpos));
        generic_set_temp_value = 0;
        }
}
```

Figure A.4: CCVS base code for writing values

```
/* Macros for storing an unsigned value into a range of bits from an integer
 * data type (byte/short/int).
 */
#define byte_set(var,startpos,bitlength,value) \
        generic_set(var,BYTE_MAX,8,startpos,bitlength,value)
#define short_set(var,startpos,bitlength,value) \
        generic_set(var,SHORT_MAX,15,startpos,bitlength,value)
#define int_set(var,startpos,bitlength,value) \
        generic_set(var,INT_MAX,31,startpos,bitlength,value)
```

Figure A.5: CCVS macros for writing unsigned values

```
/* Macros for storing a signed value into a range of bits from an integer
 * data type (byte/short/int).
 */
#define byte_set_signed(var,startpos,bitlength,value) \
        generic_set(var,BYTE_MAX,8,startpos,bitlength, \
        (value)+(1<<(bitlength-1)))
#define short_set_signed(var,startpos,bitlength,value) \
        generic_set(var,SHORT_MAX,15,startpos,bitlength, \
        (value)+(1<<(bitlength-1)))
#define int_set_signed(var,startpos,bitlength,value) \
        generic_set(var,INT_MAX,31,startpos,bitlength, \
        (value)+(1<<(bitlength-1)))
```

Figure A.6: CCVS macros for writing signed values

A.2 Usage Example

```
bool v[8];
unsigned w : 2; // w has a size of 2 bits
unsigned x : 3;
unsigned y : 4;
unsigned z : 5;
```

Figure A.7: CCVS usage example - original code

Figure A.7 shows the declaration of a few example variables. Figure A.8 shows the same variables after applying my CCVS modifications. The use of CCVS is made optional. The getter and setter macros are also defined when CCVS is disabled. These macros should be used in the rest of the model instead of accessing variable names directly. This 'indirection' allows changing how the variables are declared. There are different orders in which variables can be stored in containers. The primary goal for the programmer should be to use as few bytes of memory as possible. A secondary goal is to optimize performance. For example putting related variables in the same container could reduce the number of CPU cache misses.

```
#if ENABLE_CCVS
  byte container1
  short container2;

  #define get_v(index) get_byte (container1,index,1)
  #define get_w        get_short(container2,0,2)
  #define get_x        get_short(container2,2,3)
  #define get_y        get_short(container2,5,4)
  #define get_z        get_short(container2,9,5)

  #define set_v(index,value) set_byte (container1,index,1,value)
  #define set_w(value)       set_short(container2,0,2,value)
  #define set_x(value)       set_short(container2,2,3,value)
  #define set_y(value)       set_short(container2,5,4,value)
  #define set_z(value)       set_short(container2,9,5,value)
#else
  bool v[8];
  unsigned w : 2;
  unsigned x : 3;
  unsigned y : 4;
  unsigned z : 5;

  #define get_v(index) v[index]
  #define get_w        w
  #define get_x        x
  #define get_y        y
  #define get_z        w

  #define set_v(index,value) v[index]=value
  #define set_w(value)       w=value
  #define set_x(value)       x=value
  #define set_y(value)       y=value
  #define set_z(value)       z=value
#endif
```

Figure A.8: CCVS usage example - modified code

Appendix B

Appendix B: RAFFS Design Specification

This chapter contains a concise specification of my file store design. It describes all public and some private functions of each layer, plus all relevant variables and data structures.

B.1 Local Variables

Each process that uses the file store API must have the local variables listed below. These variables are used by the file store and its underlying layers. Other variables may also be added to storing non-shared data, such as loop indexes, temporary values and intermediary results.

fs_result Every file store API function must store its results and return codes in this variable.

page_buffer This buffer should be large enough to fit the contents of an entire Flash page. This buffer is used to store data when reading from and writing to a Flash page.

acquired_vpn Used for storing the return value of the function *ftl_acquire_unused_vpn*.

free_page_addr Used for temporarily storing a page address in the FTL.

erase_queue Used for storing a list of VPN values for locations that are pending to be erased.

erase_queue_size The number of items in the erase queue.

B.2 Locking and Concurrency

If multiple processes also allowed to concurrently access the file store API, then locking should be applied to inodes. Before a process can access an inode, it must acquire a lock on that inode. A locked inode may only be accessed by the process that locked

it. All other processes that want to access the same inode must wait until it gets unlocked and then acquire the lock for themselves. When a process wants to perform an operation that involves multiple inodes, for example removing an inode, then it should acquire locks on each of those inodes. Inodes must be locked in order of distance to the root, the closest one first. This ordering of locking prevents deadlock. When parsing a path, a lock must be acquired for each inode along that path. Once a child inode has been locked, its parent may be unlocked, provided that the parent is not needed in the file store operation that is going to be performed. A file store operation must release all acquired locks before before finishing.

A mechanism must be available to acquire a global lock. A global lock is needed for functions that require exclusive access to the whole file store. Three of such functions exist in the current specification: *fs_api_mount*, *fs_api_unmount*, and *garbage_collector*. Acquiring a global lock consists of two steps. The first step is to set the lock. Once active, other processes are not allowed to start a new operation. In the second step the process that requested the lock must wait until all active operations of other processes have completed. Once they have completed, the global lock has been acquired.

Only inodes have a locking mechanism. Other global variables are freely accessible by all processes (unless there is an active global lock). Whenever the value a global variable is modified it must be done in a single atomic step in which both the read of the current value and the write of update value take place. When searching through an array or list, this must also be done in a single atomic step. For example when looking for a free page, an unused VPN, or when reserving an inode.

B.3 Constants

There are a number of constants that control various sizes and limits in the file store and its underlying layers.

MAX_INODES The maximum number of inodes that can be present in the file store. Inodes are numbered from 0 to MAX_INODES-1.

MAX_FILE_SIZE The maximum size of a file, measured in units of data.

MAX_DIR_SIZE The maximum number of child inodes that a directory can have.

MAX_FILENAME The maximum numerical value for the name of a file or directory. The minimum value is 1. Value 0 is reserved for the root directory.

MAX_PATH_LENGTH The maximum length of a path, measured in segments.

DATA_UNIT_BITLENGTH The size of a single unit of data, measured in bits.

DATA_UNITS_PER_PAGE The number of data units that can be stored in a Flash page. Applies only when storing file data. Pages are assumed to be able to fit any inode.

NUMBER_OF_BLOCKS The number of blocks in the Flash memory.

PAGES_PER_BLOCK The number of pages in each Flash block.

NUMBER_OF_PAGES The total number of pages in the Flash memory.

RESERVED_PAGES_FOR_GC The number of free pages that are reserved for use by the garbage collection algorithm.

GC_THRESHOLD A threshold for the garbage collection algorithm. Garbage collection is skipped when the number of free pages is larger than the threshold value.

ERASE_QUEUE_MAX_SIZE The maximum size of the erase queue.

B.4 File Store API

B.4.1 Return Codes

Figure B.1 shows a list of all error codes in the file store. Every function in the file store API must return the appropriate error code in case of an error. Functions that complete without error, but have no data to return, should return *RESULT_NONE* to signal that no error was encountered. All returned data is assumed to have positive values. Negative return values are reserved specifically for the error codes. Each error code should have a distinct negative value. The value of *RESULT_NONE* is 0. When a function returns an error, no changes should have been made to the contents of the Flash memory or the state of the file store variables.

```
RESULT_NONE
RESULT_INVALID_PATH
RESULT_INVALID_OFFSET
RESULT_ALREADY_EXISTS
RESULT_INSUFFICIENT_SPACE
RESULT_EXCEEDS_MAX_SIZE
RESULT_DIR_NONEMPTY
RESULT_MAX_INODES_REACHED
RESULT_INCOMPATIBLE_INODE_TYPE
RESULT_NOT_MOUNTED
RESULT_ALREADY_MOUNTED
RESULT_ACCESS_DENIED
```

Figure B.1: Error codes

B.4.2 Paths

The files and directories that are present in the file store are structured in the form of a tree. The root directory is the root of the tree. Internal nodes in the tree are directories. A leave in the tree is either a file or a directory. All files and directories in the file store are uniquely identified with a path. A path consists of a sequence of segments. Each segment is the name of a file or directory. The path of a node is equal to the sequence of names that belong to the nodes that lie on the shortest path from the root to the node. This sequence includes the name of the node itself, and excludes the root. Names are positive numerical values. Value 0 is reserved for the root directory. All other names must be 1 or greater. Names do not need to be unique. However, all children of a directory must have different names, because paths are unique.

The numerical names can be represented by a sequence of bits. By concatenating those bit sequences, a numerical value can be obtained for a path. Path value 0 is used to identify the root. Table B.1 shows some example paths and their numerical values. Names are a 2-bit sequence in the example.

Unix path	Bit sequence (padded)	Numerical value
/	000000	0
/1	000001	1
/1/2	001001	9
/1/2/3	111001	57

Table B.1: Example path values

B.4.3 Parsing a Path

Parsing a path must be done one segment at a time. So a walk must be made in the inode tree from the root inode to the inode belonging to the path. During this walk, a lock must be acquired on each inode, starting with the root. Once a child inode has been locked, its parent must be unlocked as quickly as possible, unless that parent inode is going be modified later by the file store operation that is performing the path parsing. That situation can occur when an inode is going to be added or removed. Once a lock is acquired on a directory inode, it is safe to read the name values of its child inodes, since due to that lock it is impossible for another processes to remove any of those child inodes. The current file store API does not contain a rename function. If such a function might get added in the future, then it should keep a lock on the parent of the target inode during its operation. That should prevent any issues during path parsing done by other processes. At the end of a file store operation, all acquired locks should be released, regardless if the operation was successful or not.

B.4.4 Inode Data Structure

The file store maintains an array of inodes. The index in the array is equal to the identifier of an inode. Directory inodes must use those identifiers to refer to their child inodes. An inode is a structure with the following fields:

exists Flag that indicates whether the inode exists.

locked Flag that indicates whether the inode is locked.

reserved Flag that indicates whether the inode is reserved. A reserved inode is one that does not yet exist, but an active operation is planning to create it. Reserving an inode is required to prevent concurrent processes from creating inodes with the same identifier.

type This field indicates the type of the inode. Either a file or a directory.

name This field contains the name of the inode.

size This field contains the size of the inode.

vpn This field contains the VPN of the virtual location where the inode is stored on the storage medium.

data This field contains a list of VPN values for the data locations (in case of a file) or child inode locations (in case of a directory). For directories it also contains a list of inode identifiers for all child inodes. The size of each list is equal to the size of the inode.

B.4.5 Global Variables

Besides the inode list, the following variables are used by the file store API layer:

mounted Flag that indicates whether the file store is mounted or not.

global_lock Flag that indicates whether the global lock has been set.

active_operations The number of file store operations that is currently active. This value is used by the global locking mechanism.

B.4.6 Space

Each file store API function must reserve the amount of space that it needs prior to performing its operation. The required amount of space is equal to the number of writes that the operation will perform. The exact amount of space must be reserved that the operation is going to use.

B.4.7 fs_api_create_dir(path)

Parameters

path: The path of the directory that should be created.

Description

This functions creates a new directory that has the given path.

Order of operations

Step 1: Create inode for the new directory.

Step 2: Update inode of parent directory to reference the new child inode.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH: Path value is negative or larger than the maximum allowed value.

RESULT_ACCESS_DENIED: Path equals 0.

RESULT_INVALID_PATH: Any of the path segments between the root and the target directory does not exists or is not a directory.

RESULT_ALREADY_EXISTS: A file or directory already exists in the given path.

RESULT_EXCEEDS_MAX_SIZE: The parent directory already has its maximum allowed size.

RESULT_MAX_INODES_REACHED: The maximum allowed number of inodes already exists.

RESULT_INSUFFICIENT_SPACE: The required amount of space could not be reserved.

Required Space

2

B.4.8 fs_api_dir_exists(path)

Parameters

path: The path of the directory of which the existence should be checked.

Description

Returns true (1) if a directory exists with the given path. Returns false (0) if the directory does not exist.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

Required Space

0

B.4.9 fs_api_dir_size(path)

Parameters

path: The path of the directory for which the size should be returned.

Description

This functions checks if a directory with the given path exists and returns its size. The size of a directory is the number of child inodes that the directory has.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH: Path value is negative or larger than the maximum allowed value.

RESULT_INVALID_PATH: Any of the path segments between the root and the target directory does not exists or is not a directory.

RESULT_INVALID_PATH: The directory itself does not exist.

RESULT_INCOMPATIBLE_INODE_TYPE: The path exists but points to a file instead of a directory.

Required Space

0

B.4.10 fs_api_dir_get_child_name(path,index)

Parameters

path: The path of the directory for which the name of a child inode should be returned.

index: The index of the child inode of which the name should be returned.

Description

Returns the name of a child inode. The first child has index 0. Child nodes are stored in arbitrary order. This order may only change when a child node is added or removed.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

Required Space

0

B.4.11 fs_api_delete_dir(path)

Parameters

path: The path of the directory that should be removed.

Description

This function removes the directory with the given path. Only directories with size 0 can be removed. The root directory can not be removed.

Order of operations

Step 1: Update the parent inode to remove the reference to the directory's inode.

Step 2: Remove the inode of the directory.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH: Path value is negative or larger than the maximum allowed value.

RESULT_ACCESS_DENIED: Path equals 0.

RESULT_INVALID_PATH: Any of the path segments between the root and the target file does not exists or is not a directory.

RESULT_INVALID_PATH: The file itself does not exist.

RESULT_INCOMPATIBLE_INODE_TYPE: The path exists but points to a directory instead of a file.

RESULT_DIR_NONEMPTY: The directory has a size larger than 0.

RESULT_INSUFFICIENT_SPACE: The required amount of space could not be reserved.

Required Space

1

B.4.12 fs_api_create_file(path)

Parameters

path: The path of the file that should be created.

Description

This function creates a new file that has the given path. The file has an initial size of 0 and contains no data.

Order of operations

Step 1: Create the inode of the new file.

Step 2: Update the parent inode to refer to the new child inode.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH: Path value is negative or larger than the maximum allowed value.

RESULT_ACCESS_DENIED: Path equals 0.

RESULT_INVALID_PATH: Any of the path segments between the root and the target file does not exists or is not a directory.

RESULT_ALREADY_EXISTS: A file or directory already exists in the given path.

RESULT_EXCEEDS_MAX_SIZE: The parent directory already has its maximum allowed size.

RESULT_MAX_INODES_REACHED: The maximum allowed number of inodes

already exists.

RESULT_INSUFFICIENT_SPACE: The required amount of space could not be reserved.

Required Space

2

B.4.13 fs_api_file_exists(path)

Parameters

path: The path of the file of which the existence should be checked.

Description

Returns true (1) if a file exists with the given path. Returns false (0) if the file does not exist.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

Required Space

0

B.4.14 fs_api_file_size(path)

Parameters

path: The path of the directory for which the size should be returned.

Description

This function check if a file exists in the given path and returns its size.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH Path value is negative or larger than the maximum allowed value.

RESULT_INVALID_PATH Any of the path segments between the root and the target file does not exists or is not a directory.

RESULT_INVALID_PATH The file itself does not exist.

RESULT_INCOMPATIBLE_INODE_TYPE The path exists but points to a directory instead of a file.

Required Space

0

B.4.15 fs_api_file_append(path,data)

Parameters

path: The path of the file to which a unit of data should be appended.

data: The value of the data unit that should be appended.

Description

This function appends one unit of data to the end of the file. The size of the file increases with one.

Order of operations

Step 1: If the last data location that is currently occupied with data from the file has room for another unit of data, then read the contents of that data location, append

the new value, and write the updated data to a new data location. Otherwise, write the unit of data to a new data location.

Step 2: Update the file's inode with the new size. Add or update the data location from step 1.

Step 3: Erase the old data location from step 1, if any.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH: Path value is negative or larger than the maximum allowed value.

RESULT_INVALID_PATH: Any of the path segments between the root and the target file does not exists or is not a directory.

RESULT_INVALID_PATH: The file itself does not exist.

RESULT_INCOMPATIBLE_INODE_TYPE: The path exists but points to a directory instead of a file.

RESULT_EXCEEDS_MAX_SIZE: The current size of the file is already equal to the maximum allowed size.

RESULT_INSUFFICIENT_SPACE: The required amount of space could not be reserved.

Required Space

2

B.4.16 fs_api_file_modify(path,offset,data)

Parameters

path: The path of the file in which the value of a data unit must be modified.

offset: The offset of the desired data unit within the file.

data: The new value of the data unit.

Description

This function modifies the value of the data unit at the given offset in the file with the given path. The first data unit of a file has offset 0.

Order of operations

Step 1: Read the content of the data location that contains the current value of the target data unit. Update the value and write the data to a new data location.

Step 2: Update the file's inode to reference the new data location instead of the old one.

Step 3: Erase the old data location.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH: Path value is negative or larger than the maximum allowed value.

RESULT_INVALID_PATH: Any of the path segments between the root and the target file does not exists or is not a directory.

RESULT_INVALID_PATH: The file itself does not exist.

RESULT_INCOMPATIBLE_INODE_TYPE: The path exists but points to a directory instead of a file.

RESULT_INVALID_OFFSET: The offset is equal to or larger than the size of the file.

RESULT_INSUFFICIENT_SPACE: The required amount of space could not be reserved.

Required Space

1

B.4.17 fs_api_file_read(path,offset)

Parameters

path: The path of the file from which the value of a data unit must be read.

offset: The offset of the desired data unit within the file.

Description

This function reads the value of the unit of data at the given offset in the file with the given path. The first data unit has offset 0.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH: Path value is negative or larger than the maximum allowed value.

RESULT_INVALID_PATH: Any of the path segments between the root and the target file does not exists or is not a directory.

RESULT_INVALID_PATH: The file itself does not exist.

RESULT_INCOMPATIBLE_INODE_TYPE: The path exists but points to a directory instead of a file.

RESULT_INVALID_OFFSET: The given offset is equal to or greater than the size of the file.

Required Space

0

B.4.18 fs_api_file_truncate(path,size)

Parameters

path: The path of the file that should be truncated.

size: The desired size of the file after truncation.

Description

This function truncates a file to the given size. All data units with an offset equal to or larger than the target size are removed. The first data unit has offset 0.

Order of operations

Step 1: If a data location exists that contains multiple data units of the file, of which only some data units need to be removed, then read this data location. Remove the data units that are no longer needed, and write the data to a new location. This situation can only occur if the target size is larger then 0 and not an exact multiple of *DATA_UNITS_PER_PAGE*.

Step 2: Update the file's inode with the new size. Also update the reference to the data location that was changed in step 1.

Step 3: Erase data locations that contain all the remaining data units that must be removed. Also erase the old data location from step 1.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH: Path value is negative or larger than the maximum allowed value.

RESULT_INVALID_PATH: Any of the path segments between the root and the target file does not exists or is not a directory.

RESULT_INVALID_PATH: The file itself does not exist.

RESULT_INCOMPATIBLE_INODE_TYPE: The path exists but points to a directory instead of a file.

RESULT_INVALID_OFFSET: The parameter size is greater than the size of the file.

RESULT_INSUFFICIENT_SPACE: The required amount of space could not be reserved.

Required Space

1 if the target size is 0 or a multiple of *DATA_UNITS_PER_PAGE*, 2 otherwise.

B.4.19 fs_api_delete_file(path)

Parameters

path: The path of the file that should be removed.

Description

This function removes the file that has the given path.

Order of operations

Step 1: Update the parent inode to remove the reference to the file's inode.

Step 2: Remove the inode of the file.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

RESULT_INVALID_PATH: Path value is negative or larger than the maximum allowed value.

RESULT_ACCESS_DENIED: Path equals 0.

RESULT_INVALID_PATH: Any of the path segments between the root and the target file does not exists or is not a directory.

RESULT_INVALID_PATH: The file itself does not exist.

RESULT_INCOMPATIBLE_INODE_TYPE: The path exists but points to a directory instead of a file.

RESULT_DIR_NONEMPTY: The directory has a size larger than 0.

RESULT_INSUFFICIENT_SPACE: The required amount of space could not be reserved.

Required Space

1

B.4.20 fs_api_mount()

Description

This function mounts the file store. The mount algorithm is described in section 6.1.3. This function requires a global lock.

Error conditions

RESULT_ALREADY_MOUNTED: File store is already mounted.

Required Space

0

B.4.21 fs_api_unmount()

Description

This function deallocates all inodes that are stored in RAM, resets all file store variables, resets all FTL variables, and sets the mount state to *unmounted*. This function requires a global lock.

Error conditions

RESULT_NOT_MOUNTED: File store is not mounted.

Required Space

0

B.5 Flash Translation Layer

B.5.1 Global Variables

vpn_mapping_table[] Table for mapping VPN values to physical page addresses. The rows of the table correspond to the VPN values. Each rows in the table should contain the following fields: *valid*, *page_address*, and *mountdata*. The field *valid* indicates whether the entry is a valid VPN value that is in use by the file store. The field *page_address* contains the physical address of the page belonging to a VPN. The field *mountdata* is used only during mounting, namely to temporarily store page versions and a reference flag.

free_pages The amount of pages in the Flash memory with state *free*.

dirty_pages The amount of pages in the Flash memory with state *obsolete*.

reserved_free_pages The amount of pages that has been reserved.

free_page_list[] Bitvector that indicates which physical pages are free.

block_dirty_page_counts[] List of dirty page counts for each block in the Flash memory.

B.5.2 ftl_acquire_unused_vpn()

Description

Selects an unused Virtual Page Number and marks it as valid in the VPN table. A free physical page must be selected and the address of that page will be stored in the FTL mapping table with as index the selected VPN. The amount of reserved free space must be decremented after selecting the free page. The value of the selected VPN is stored in the local variable *acquired_vpn*. All of the above actions must be performed in a single atomic step.

Preconditions

The amount of reserved free must be at least 1.

The local variables *acquired_vpn* and *free_page_addr* have value 0. This indicates that these variables don't contain any valid information.

Postconditions

The local variable *free_page_addr* has value 0.

The local variable *acquired_vpn* contains the value of the selected VPN.

B.5.3 ftl_invalidate_page(page_addr)

Description

Invalidates the contents of the page with the address given by the parameter *page_addr*. This is done by programming the spare data region of the page. All bits in the programmed data must should be set to 1, except of the *valid_obsolete* status bit. The amount of dirty space must be incremented. The dirty page count of the block to which the affected page belongs must be incremented. This function may only be called from within the FTL layer.

Preconditions

The current status of the page must be *valid*.

B.5.4 ftl_write_pagebuffer_to_flash(target_addr)

Description

This functions programs the contents of the *page_buffer* to the Flash page specified by parameter *target_addr*. Programming must be done using a two step protocol. In the first step the status field in the page buffer is set to *invalid* and the page is programmed. In the second step only the spare data region of the page must be programmed. The only change in this second step must be setting the status field value to *valid*. This function may only be called from within the FTL layer.

Preconditions

The target page has state *free*.

Postconditions

The target page has state *valid*.

B.5.5 ftl_write_data(vpn)

Description

This function writes the contents of the page buffer to Flash. The logical target location is given by the parameter *vpn*. This value should have a valid entry in the FTL mapping table. The spare data region of the page buffer must be overwritten with the contents of the physical page associated with the given VPN. After that there are two distinct situations, based on the contents of that physical page. Situation one is when the physical page already contains data, having state *valid*. In that case the contents of the page buffer must be written to a free page. Before that is done, the version field in the page buffer must be incremented, followed by a modulo 4 operation. After the page has been programmed, the FTL mapping table must updated with the new location of the data. The page that contains the old copy of the data must be invalidated once the new data has been stored successfully. Situation two is when the physical page has state *free*. In that case this page must be used to store the contents of the page buffer. Before the page is programmed, the VPN value must be set in the spare data region of the page buffer. The version field must be set to value 0.

Preconditions

The value of parameter *vpn* has a valid entry in the FTL mapping table.

The local variable *free_page_addr* has value 0.

Postconditions

The local variable *free_page_addr* has value 0.

B.5.6 ftl_read_data(vpn)

Description

Reads the contents of the page with the logical address given by the parameter *vpn*. This VPN must have a valid entry in the FTL mapping table. The contents of the read page is stored in the local variable *page_buffer*.

Preconditions

The given VPN must have a valid entry in the FTL mapping table.

The physical page associated with the given VPN should have state *valid*.

The VPN value stored in the spare data region of the physical page must be identical to the value of the parameter *vpn*.

B.5.7 ftl_erase_data(vpn)

Description

Invalidates the page associated with the logical address given by the parameter *vpn*. This VPN must have a valid entry in the FTL mapping table. The VPN mapping table must be updated so that the given VPN is no longer valid.

Preconditions

The given VPN must have a valid entry in the FTL mapping table.

The physical page associated with the given VPN should have state *valid*.

Postconditions

The given VPN should be an invalid entry in the FTL mapping table.

The physical page associated with the given VPN should have state *obsolete*.

B.5.8 ftl_add_to_erase_queue(vpn)

Description

Adds the given VPN value to the end of the erase queue. The data location belonging to that VPN will be erased when the function *ftl_process_erase_queue()* is called.

Preconditions

The erase queue has a size smaller than the value of constant *ERASE_QUEUE_MAX_SIZE*.

B.5.9 ftl_process_erase_queue()

Description

Erases all data locations that are listed in the erase queue, by calling *ftl_erase_data()* for each entry. The entries do not have to be processed in any specific order.

Postconditions

The erase queue is empty.

B.5.10 ftl_init_mapping_table()

Description

Reads the spare data region of every single page in the Flash memory. The gathered information is used to set the values of the mapping table and other variables in the FTL.

The information found in pages with state *valid* must be stored in the mapping table. The index in the mapping table is equal to the VPN value stored in the page. The field *valid* must be set to 1. The field *page_address* must be set to the physical address of the page. The field *mountdata* must be set to the value of the version of the page. If a page is encountered with a VPN value that already has a valid entry in the mapping table, then the data of the page with the newest version must be stored in the mapping table, and the page with the older version must be marked as obsolete by calling the function *ftl_invalidate_page*. Version X is newer than version Y if: $X = (Y + 1) mod(4)$.

If a page is encountered with state *free*, then that page must be marked as free in the free page list. The amount of free pages must also be incremented.

If a page is encountered with state *invalid*, then that page must be marked as obsolete by calling the function *ftl_invalidate_page*.

If a page is encountered with state *obsolete*, then the dirty page count must be incremented. The dirty page count for the block to which the page belongs must also be incremented.

Once all pages have been processed, the value of every *mountdata* field in the mapping table must be reset to 0.

Preconditions

The mapping table contains no valid entries.

The free page, dirty page, and block dirty page counts all have value 0.

There are no pages listed as free in the free page list.

Postconditions

The *mountdata* field for each entry in the mapping table has value 0.

B.5.11 ftl_mark_vpn_as_referenced(vpn)

Description

Marks the given VPN as referenced in the mapping table. This is done by setting the value of the *mountdata* field to 1. This function may only be called by the function *fs_api_mount()*.

Preconditions

The given VPN corresponds to a valid entry in the mapping table.

The value of the *mountdata* field for the given VPN equals 0.

B.5.12 ftl_remove_unreferenced_vpn()

Description

Erases all valid VPN entries found the mapping table that have not been marked as referenced. A VPN value is marked as referenced when its *mountdata* field has value 1. Once a VPN entry has been erased, it must be marked as invalid in the mapping table, setting all fields to 0 values for that row in the mapping table. Once all entries in

the table have been processed, the value of every *mountdata* field in the mapping table must be reset to 0. This function may only be called by the function *fs_api_mount()*.

B.5.13 garbage_collector()

Description

Perform garbage collection. The algorithm must work as follows:

1. If the number of free page is below the threshold value *GC_THRESHOLD*, then continue, otherwise quit.

2. Find the block with the highest dirty page count. If multiple exist with the same highest count, then any of those may be picked.

3. Read the spare data region of each page in this block. For each page with state *valid*, copy its contents to a free page in another block. The version field in the spare data region must be incremented (module 4). The amount of reserved space must *not* be decremented, because free space is implicitly reserved specifically for the garbage collector. Once the data has been copied, the page that contained the original copy must be marked as obsolete by calling the function *ftl_invalidate_page()*.

4. Erase the block.

5. Go back to step 1.

B.5.14 ftl_reserve_space(amount)

Description

Adds the requested amount to the current amount of reserved space. This must be done only when there is a sufficient amount of free space. There is sufficient free space when the amount of free pages is equal to or greater than the current amount of reserved pages plus the requested amount plus *RESERVED_PAGES_FOR_GC*. If there is insufficient free space then the amount of reserved space must remain unchanged, and the value of the local variable *fs_result* must be set to *RESULT_INSUFFICIENT_SPACE*. The above steps must be done in a single atomic block. If *fs_result* equals an error code when this function is called, this function must be skipped.

B.5.15 ftl_get_free_page(reserved,block_number)

Description

This function is used to acquire the address of a free page. The address must be stored in the local variable *free_page_addr*. The first available free page is selected. Acquiring a free page implies that the page will get programmed. Any code that requests a free page, but never programs it is faulty. The parameter *reserved* indicates whether the space was reserved. If that is the case then the amount of reserved space must be decremented. The parameter *block_number* contains the index of a block. The selected free page may not belong to that block. This block exclusion must only done when the parameter *reserved* has value false. Only the garbage collector may call this

function with *reserved* set to false. The above steps must be done in a single atomic block. This function may only be called from within the FTL.

Preconditions

If *reserved* is true, then the amount of reserved space must be greater than or equal to 1.

B.6 Flash Driver

B.6.1 flash_api_erase_block(block_number)

Description

Erases the block with the index given by parameter *block_number*. Erasing means resetting all bits of every page that belongs to the block to logical value 1.

Preconditions

All pages in the block must have either state *free* or state *obsolete*.

B.6.2 flash_api_read(page_number)

Description

Reads the contents of the page with address given by parameter *page_number*. The read data must be stored in the local variable *page_buffer*, overwriting any previous contents.

B.6.3 flash_api_read_spare_region(page_number)

Description

Reads the contents of the spare data region of the page with index given by parameter *page_number*. The read data must be stored in the local variable *page_buffer*, overwriting any previous contents. Only the spare data region of the page buffer may be overwritten. The contents of the data region may not be modified in any way.

B.6.4 flash_api_program(page_number)

Description

Programs the contents of the page buffer into the page with index given by parameter *page_number*. The new contents of the Flash page must be the result of a bitwise AND operation of the contents of the page buffer and the current contents of the Flash page.

B.6.5 flash_api_program_spare_region(page_number)

Description

Programs the contents of only the spare data region of the page buffer into the spare data region of the page with index given by parameter *page_number*. The new contents of the Flash page must be the result of a bitwise AND operation of the contents of the page buffer and the current contents of the Flash page.

B.7 Flash Memory

The Flash memory consists of an array of pages, declared as a global variable. The first page has index 0. The first block has index 0. Pages belonging to the same block are grouped together. Page 0 belongs to block 0. The block number of a page can be calculated by dividing the page address by *PAGES_PER_BLOCK*.

B.7.1 Flash Page Structure

A page is a data structure with the following fields, of which the first three belong to the spare data region:

state This field contains a 3-bit value for the page state.

vpn VPN value.

version A 2-bit version field.

data Used to store inode data or file data. This field must be large enough to store the largest possible inode. The inode data fields that needs to be stored on the Flash memory are: id, type, name, size, and the VPN values of data locations or child inodes.

B.7.2 Page States

Allowed page states are:

- 111: Free
- 011: Invalid
- 010: Valid
- 000: Obsolete