Formal Modeling and Analysis of a Flash Filesystem in Alloy

Eunsuk Kang and Daniel Jackson

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA. U.S.A.
{eskang,dnj}@mit.edu

Abstract. This paper describes the formal modeling and analysis of a design for a flash-based filesystem in Alloy. We model the basic operations of a filesystem as well as features that are crucial to NAND flash hardware, such as wear-leveling and erase-unit reclamation. In addition, we address the issue of fault tolerance by modeling a mechanism for recovery from interrupted filesystem operations due to unexpected power loss. We analyze the correctness of our flash filesystem model by checking trace inclusion against a POSIX-compliant abstract filesystem, in which a file is modeled simply as an array of data elements. The analysis is fully automatic and complete within a finite scope.

1 Introduction

Flash memory is becoming an increasingly popular choice of medium for non-volatile data storage. Among a wide range of applications, flash memory has been used by NASA for on-board storage in planetary rovers. In one well known incident, *Spirit*, a Mars Exploration Rover, suffered a major system failure that resulted in 10 days of lost scientific activity [19]. The cause of the failure was later determined to be a flaw in the flash filesystem software. On investigation, it turned out that the failure scenario was neglected during testing because the development team considered it to be an "unanticipated behaviour."

Testing is an essential part of any software development process, but cannot alone ensure the reliability of software. Formal methods can mitigate the weakness of testing by allowing an exhaustive analysis. However, applying formal methods to a poorly designed piece of code in an after-the-fact, ad hoc fashion is impractical, and rarely yields high confidence for reliability. Instead, by formalizing important aspects of a design and analyzing them early in a development process, software engineers can identify key reliability issues and address them in the simpler context of the design, where they can be resolved before the complexities of implementation are introduced.

This paper describes the formal modeling and analysis of a design for a flash filesystem in Alloy [16]. Our model addresses three primary aspects of a flash filesystem: (1) The underlying flash hardware, (2) filesystem software with basic file operations such as read and write, and (3) a fault-tolerance mechanism for handling unexpected hardware failures. Unlike disk-based storage devices, flash memory suffers from a major limitation in that blocks can be written only a limited number of times. In order to address this issue, our model includes techniques for efficiently managing block erasures, such as wear-leveling and erase-unit reclamation [10].

Alloy provides completely automatic (but bounded) analysis of a model, given a property and a scope for the size of each domain in the model. In order to verify the correctness of the filesystem design, we used a simplified version of the POSIX standard as a reference model. Then, we checked trace inclusion of the concrete flash filesystem against the reference model using backwards simulation [24]. To our knowledge, this is the first fully automatic analysis of a design for a flash filesystem.

The paper presents the POSIX reference model (Section 2), a model of the underlying flash hardware (3), and then a design of a flash filesystem that uses the flash hardware and is intended to conform to the reference model (4). Unlike the first two models, this design does not formalize existing descriptions, but it incorporates a variety of mechanisms that have appeared in the literature [10, 15]. Subsequent sections describe the analysis that was performed (5), relate our approach to others (6), and discuss the challenges faced during this project (7) and plans for future work (8).

2 Abstract POSIX Filesystem

POSIX (Portable Operating System Interface) [21] is an international standard that specifies the function signatures and expected behaviours of a set of filesystem operations. The widespread adoption of POSIX by many popular operating systems, such as UNIX and Mac OS X, makes it an attractive choice as a reference model for a flash filesystem. A specification for a UNIX filesystem was formalized in the Z notation [20] by Morgan et al. [18] and served as a starting point for our work.

Alloy [16] is a declarative modeling language based on first-order logic with transitive closure. The origin of Alloy is rooted in Z, drawing on the latter's simple and intuitive semantics that is well suited for object and data modeling. For example, a file with an array of data elements can be declared as follows[1]:

```
sig File { contents : seq Data }
sig Data {}
```

Then, an entire filesystem may be viewed as a container for a relation that maps each file identifier to zero or one file[2]:

```
sig AbsFsys { fileMap : FID -> lone File }
sig FID {}   // File identifier
```

[1] In Alloy, the keyword `sig S` declares a set of atoms of type `S`, and `seq T` constructs a sequence whose elements are of type `T`.

[2] The keyword `lone` imposes a cardinality constraint of "less than or equal to one".

The basic operations of the filesystem—reading from and writing to a file—are modeled using functions and predicates. The read operation, as defined by POSIX, takes three arguments—a file identifier, an offset, and a size—and returns a data sequence of the specified size, starting at the offset within the file:

```
fun readAbs[fsys: AbsFsys, fid: FID, offset, size: Int]: seq Data {
   let file = fsys.fileMap[fid] |
       // subseq[m,n] returns a subsequence between m and n, inclusively
       (file.contents).subseq[offset, offset + size - 1]
}
```

Like Z, Alloy does not support the notion of an implicit global state. Thus, `readAbs` explicitly takes an instance of `AbsFsys` as an argument.

The write operation takes a file identifier, an offset, a size, and a buffer containing the input data, and writes a data sequence of the specified size from the buffer into the file, starting at the offset[3]:

```
pred writeAbs[fsys, fsys': AbsFsys, fid: FID, buffer: seq Data,
              offset, size: Int] {
   let buffer' = buffer.subseq[0, size - 1],
       file = fsys.fileMap[fid], file' = fsys'.fileMap[fid] {
          (#buffer' = 0) => file' = file
          (#buffer' != 0) =>
             file'.contents =
                (zeros[offset] ++ file.contents) ++ shift[buffer', offset]
          promote[fsys, fsys', file, file', fid]
       }
}
```

Sequences are represented, as in Z, by functions from integers to values. The expression `zeros[n]` gives a sequence of zeros of length `n`, and is used for padding; `shift[s,i]` gives a function like the sequence `s` but with the index of each element shifted by `i`. Unlike `readAbs`, `writeAbs` is expressed as a predicate (rather than a function) because it may modify the state of the filesystem.

There are three distinct cases to consider in this operation. First, if the input buffer is empty, `writeAbs` does not modify the contents of the file. If the offset is located within the file, `writeAbs` overrides the existing data in the overlapping positions with the input data. Lastly, if the offset is greater than the file size, `writeAbs` fills the gap between the end of the file and the offset with zeros.

Changes in the state of the filesystem are modeled using an explicit constraint between a pair of pre- and post- states (`fsys` and `fsys'`). It is also necessary to ensure that the operation does not affect any other files. This style of modeling changes in system state is called *promotion* [24]. The following predicate "promotes" a change in the contents of a file to a change in the entire filesystem:

```
pred promote[fsys, fsys': AbsFsys, file, file': File, fid: FID] {
   file = fsys.fileMap[fid]
   fsys'.fileMap = fsys.fileMap ++ (fid -> file')
}
```

[3] The cardinality operator `#` returns the length of a sequence, and `++` is the operator for relational override.

3 NAND Flash Memory

Two types of flash memory are currently in widespread use: NOR and NAND. Although NOR allows random access and is easier to program, the higher density and performance of NAND makes the latter more suitable as a storage device. Our model of the flash hardware is based on Open NAND Flash Interface (ONFi), an industry-wide standard for the specification of NAND flash memory [14]. However, it is important to note that the focus of our work is on the design of a filesystem, not a flash device. Therefore, our hardware model includes only the minimum amount of detail necessary for modeling basic flash operations, erase-unit reclamation, and fault tolerance.

3.1 Memory Hierarchy

The smallest unit for reading or programming flash memory is called a *page*; it consists of a fixed number of data elements[4]:

```
sig Page { data : seq Data } { #data = PAGE_SIZE }
```

In addition, each page is associated with one of four status constants:

```
abstract sig PageStatus {}
one sig Free,                          // Erased and ready to be programmed
        Allocated,                     // Allocated for a file write operation
        Valid,                         // Contains valid data in a file
        Invalid extends PageStatus {}  // Contains obsolete data
```

A *block*, also called an *erase-unit*, contains an array of pages and is the smallest unit for erase operations. A *logical unit* (LUN) is the minimum independent entity that receives and executes a flash command[5]:

```
sig Block { pages : seq Page } { #pages = BLOCK_SIZE }
sig LUN { blocks : seq Block } { #blocks = LUN_SIZE }
```

Two forms of addresses are used during flash operations: the *row address* and the *column address.* A row address is used to access a particular page:

```
sig RowAddr { lunIndex, blockIndex, pageIndex : Int }
```

A column address, simply of type `Int`, identifies the position of a data element within a page.

Finally, a flash *device* is the top-level component that directly communicates with the host filesystem:

[4] A formula `F` in `sig A {...}{F}` is a constraint that applies to every atom of type `A`.
[5] ONFi defines another level of hierarchy—called *targets*—above LUNs. To simplify our analysis, we abstract away this detail from our model.

```
sig Device {
   luns : seq LUN,
   pageStatusMap : RowAddr -> one PageStatus,
   eraseCountMap : RowAddr -> one Int,
   reserveBlock : RowAddr
}{ #luns = DEVICE_SIZE }
```

The three fields `pageStatusMap`, `eraseCountMap`, and `reserveBlock` are auxiliary data structures used for erase-unit reclamation and fault tolerance[6]. The `pageStatusMap` associates each page in the device with its current status. The `eraseCountMap` associates each block with the number of times it has been erased; erase counts play a crucial role in wear-leveling. Lastly, `reserveBlock` holds the address of a block that temporarily stores valid pages during erase-unit reclamation. The usage of these data structures is further discussed in Section 4.

3.2 Flash API Functions

During a file operation, the host filesystem may make one or more calls to three flash API functions: read, program, and erase. Due to limited space, we present only the interface declarations of these operations:

```
// Reads data from the page at "rowAddr", starting at the index "colAddr"
fun fRead[d: Device, colAddr: Int, rowAddr: RowAddr] : seq Data { ... }

// Programs (i.e. writes) "newData" into the page at "rowAddr", starting at
// the index "colAddr", and sets the status of the page to "Allocated"
pred fProgram[d,d': Device, colAddr: Int, rowAddr: RowAddr,
              newData: seq Data] { ... }

// Erases the entire block that contains the page at "rowAddr", increments
// its erase count, and sets the status of every page within the block to "Free"
pred fErase[d,d': Device, rowAddr: RowAddr] { ... }
```

Note that `fProgram` and `fErase` are expressed as constraints between two device states (`d` and `d'`) since these operations may modify the state of the device.

4 Flash Filesystem

Given the model for the underlying hardware, we now describe a concrete filesystem that communicates with the flash device to perform file operations. This concrete model is not based on one particular flash filesystem; rather, our design incorporates a variety of mechanisms that have appeared in literature. Namely, we adopted the techniques for wear-leveling and erase-unit reclamation from Gal and Toledo's survey paper on flash memory algorithms [10]. The division of the write operation into separate phases and the mechanism for power-loss recovery

[6] ONFi does not explicitly mention these data structures. On an actual device, they would be scattered across the flash memory using sophisticated techniques, but this detail is not suitable for the level of modeling abstraction in this work.

were modeled after the Intel Flash File System Specification [15]. It is important to note that in our modeling task, we were primarily concerned with the correctness of the design, not its performance. Thus, when multiple techniques were available, we adopted the alternative that we considered to be the simplest.

A file, represented by an `Inode`, consists of a list of virtual blocks (`VBlock`), each of which points to a particular page on the flash device:

```
sig Inode { blockList : seq VBlock }
sig VBlock {}
```

Like its abstract counterpart, the concrete filesystem contains a relation that maps each file identifier to at most one inode. In addition, the filesystem contains a bijective map from a virtual block to a row address:

```
sig ConcFsys {
   inodeMap : FID -> lone Inode,
   blockMap : VBlock one -> one RowAddr
}
```

Rather than being a fixed map, `blockMap` is dynamically updated during write operations, and plays a crucial role in wear-leveling, as discussed below.

4.1 Concrete Operations

The two basic file operations that we describe here—read and write—are substantially more complex than their counterparts in the abstract filesystem. A concrete operation involves multiple calls to the flash API functions, since an inode consists of a number of fixed-size pages. Due to limited space, we focus on the most distinctive aspects of the operations.

Concrete Read Like `readAbs`, the `readConc` operation (Fig. 1) accepts three arguments—`fid`, `offset`, and `size`. In addition, `readConc` requires a `ConcFsys` and a `Device`, which together represent a particular state of the filesystem. Note that unlike `readAbs`, `readConc` is a predicate (not a function) that constrains `buffer` to be the result of the operation[7].

The core part of `readConc` (shown in Fig. 1) involves reading each of the virtual blocks in the inode and storing the output into a single, contiguous buffer. Prior to line 4, `blocksToRead` is constrained to be a sequence of virtual blocks to be read, based on `offset` and `size`. For each index `i` in this sequence, `readConc` retrieves the address of the page to which the virtual block at `i` is mapped (line 6), invokes `fRead` (line 9), and stores the page data into a buffer slot between two indices, `from` and `to`. After line 9, `buffer` contains all of the data from `blocksToRead` in the order that they appear within the inode.

[7] Sometimes, it is more natural to describe a result implicitly rather than to construct it explicitly using a function with the formula as the body of a set comprehension.

```
pred readConc[fsys: ConcFsys, d: Device, fid: FID, offset, size: Int,
              buffer: seq Data] {
  ...
  all idx : blocksToRead.inds |   // "inds" returns the set of all indices
    let vblock = blocksToRead[idx],
        rowAddr = fsys.blockMap[vblock],
        from = PAGE_SIZE * idx, to = from + PAGE_SIZE - 1 |
          // Read a flash page and store data into correct buffer slot
          buffer.subseq[from,to] = fRead[d, 0, rowAddr]
  ...
} // 90 LOC in total, including comments
```

```
pred writeConc[fsys, fsys': ConcFsys, d, d': Device, fid: FID,
               buffer: seq Data, offset, size: Int] {
  ...
  some stateSeq : seq TranscState, interDev : Device {
    // Phase 1: Program pages
    stateSeqConds[d, interDev, stateSeq, numPagesToProgram]
    all idx : stateSeq.butlast.inds {
      let inode = fsys.inodeMap[fid],
          from = PAGE_SIZE * idx, to = from + PAGE_SIZE - 1,
          dataFragment = buffer.subseq[from, to],
          vblock = inode.blockList[startBlkIndex + idx],
          rowAddr = fsys.blockMap[vblock]
          preState = stateSeq[idx], postState = stateSeq[idx + 1] |
            // Program one page worth of data into the flash
            programVBlock[preState, postState, rowAddr, dataFragment]
    }
    // Phase 2: Invalidate/validate old/new pages
    updatePageStatuses[interDev, d']
    // Update virtual-block-to-page mapping and the list of inode blocks
    updateFsysInfo[fsys, fsys', fid, stateSeq.last]
  }
  ...
} // 549 LOC in total, including comments
```

Fig. 1. Concrete Operations

Wear-Leveling and Erase-Unit Reclamation Unlike sectors in a traditional disk-based filesystem, flash pages must be completely erased before they can be rewritten. One major limitation of flash memory is that each block can be erased only a finite number of times. Thus, a *wear-leveling* technique that distributes erasures evenly across flash memory is essential in any flash filesystem.

Using an example, let us illustrate the standard wear-leveling technique [10] that is adopted by most flash filesystem, including our design. Suppose an inode `n` consists of a list of virtual blocks, one of which (`vblk`) is mapped to a physical flash page `p1`. A client sends a request to the filesystem to overwrite the existing data, including `vblk`, in the inode. A simple approach would be to erase the physical flash block `fblk` that contains `p1` and then program new data into `p1`. However, if operations involving `n` were frequent, then `fblk` would wear out much more quickly than others. Thus, rather than erasing `p1`, we instead program the new data into a free, available page `p2`. In addition, we mark the data in `p1` as obsolete by modifying the page status to `Invalid`. Lastly, we update `blockMap` in `ConcFsys` to indicate that `vblk` is now mapped to `p2`.

Over time, the flash device accumulates obsolete data and eventually runs out of free pages. In order to free up space for new program operations, the filesystem carries out a procedure called *erase-unit reclamation*. A reclamation procedure involves the following steps:

1. Search for all blocks that contain obsolete data. Among these blocks, select the one with the lowest erase count by checking `eraseCountMap` in `Device`.
2. Relocate all valid pages in the selected block. This block (call it `dirtyBlock`) may still contain one or more pages that hold valid data. For the purpose of relocation, the filesystem keeps one completely erased block as a spare (`reserveBlock` in `Device`). Each valid page is relocated from `dirtyBlock` to the corresponding position in `reserveBlock`.
3. Erase `dirtyBlock` using the `fErase` command. This block becomes the new reserve block for the filesystem.

After Step 3, all pages within the old `reserveBlock` that do not hold the relocated data from `dirtyBlock` are free and available for programming.

Concrete Write The `writeConc` operation (Fig. 1) is expressed as a constraint between two pairs of `ConcFsys` and `Device` atoms. The pair (`fsys, d`) represents the state of the filesystem at the beginning of the operation, and (`fsys', d'`) represents the state at the end. At the filesystem client level, `writeConc` is a single-step transition between `fsys` and `fsys'`, modifying the filesystem in the following ways: 1) If `writeConc` involves overwriting existing data in the inode, then it updates `fsys.blockMap` with a new virtual-block-to-page mapping, and 2) if `writeConc` involves writing data beyond the current end of the inode, then it appends one or more virtual blocks to `inode.blockList`.

At the flash device level, `writeConc` makes a sequence of calls to the flash command `fProgram`, depending on the size of the input data and `PAGE_SIZE`. In order to model the flash operations closely to ONFi, we explicitly introduce a sequence of intermediate device states between `d` and `d'`; each pair of adjacent states in this sequence corresponds to a pair of pre- and post- states that are passed as arguments to `fProgram`. After each step along the sequence, `writeConc` maintains information about allocated-to-obsolete-page pairs and a list of new pages to be added to the inode; this auxiliary information is used to update `fsys.blockMap` and `inode.blockList` at the end of the operation. The signature `TranscState` encapsulates all of the stateful information:

```
sig TranscState {
  dev : Device,                // Current device state
  allocToObsoletePagePairs : RowAddr -> lone RowAddr, // New-old page pairs
  newPageList : seq RowAddr   // List of new pages to be added to inode
}
```

Then, we can model a sequence of flash-level transitions using a sequence of `TranscState`'s, with additional constraints as follows:

```
pred stateSeqConds[init, final: Device, stateSeq: seq TranscState, length: Int]{
   stateSeq.first.dev = init    // Beginning of trace is initial device state
   stateSeq.last.dev = final    // End of trace is final device state
   #stateSeq = length + 1       // Constrain the length of sequence
   no stateSeq.first.allocToObsoletePagePairs  // Initially empty pairs
   no stateSeq.first.newPageList               // Initially empty list
}
```

Fig. 1 shows a core snippet from a simplified version of the full `writeConc` model. Based on the Intel specification (Section 2.5) [15], we divide the operation into two distinct phases. Phase 1 (lines 6-16) involves partitioning the input buffer into fixed-size fragments and programming them into the flash memory. For each `i`, which corresponds to the $\texttt{i}^{th}$ transition in `stateSeq`, `writeConc` extracts a data fragment of length `PAGE_SIZE` from the buffer (lines 9-10). Next, `writeConc` retrieves the row address of the virtual block that will be overwritten with this fragment (lines 11-12). Finally, in line 15, the predicate `programVBlock` programs the data fragment into the virtual block (which will be mapped to a new flash page) by executing `fProgram` and adds a (new, old) row address pair to `preState.allocToObsoletePagePairs`.

If the expression `(startBlkIndex + i)` evaluates to an index beyond the end of `inode.blockList`, then both `vblock` and `rowAddr` will be empty expressions (lines 11 and 12). The predicate `programVBlock` handles this case by appending a page to `preState.newPageList`. In addition, if the device is out of free pages, `programVBlock` performs erase-reclamation before programming a page.

In Phase 2, we invalidate the pages that contain obsolete data and then validate all of the pages that were allocated during Phase 1; for simplicity, we present this phase as being carried out inside the predicate `updatePageStatuses` (line 18). The quantified variable `interDev`, introduced in line 4, acts as an intermediate device state that joins the two phases together.

Lastly, after all of the flash-level transitions have been completed, `writeConc` updates `fsys.blockMap` and `inode.blockList` using the information accumulated up to the last `TranscState` in the state sequence (line 20).

4.2 Fault Tolerance Mechanism

Over the course of its lifetime, a flash device is susceptible to a variety of unexpected hardware failures. Therefore, one crucial aspect of designing a flash filesystem is its robustness in recovering from such failures. After recovery, the filesystem must be either in a state as if an operation has never begun, or in a state where the operation has been successfully completed. In this work, we modeled one particular type of fault-tolerance mechanism—recovery from power loss in the middle of a write operation. Our model is based on the mechanism that is described in the Intel specification (Section 2.5) [15].

A power failure can occur during either Phase 1 or Phase 2 of the write operation. We give a high-level description of the fault-tolerance mechanism in

```
pred alpha[asys: AbsFsys, csys: ConcFsys, d: Device] {
  all fid : FID |
  let file = asys.fileMap[fid], inode = csys.inodeMap[fid],
      vblocks = inode.blockList {
         #file.contents = #vblocks * PAGE_SIZE
         (all i : vblocks.inds |
          let vblock = vblocks[i],
              from = i * PAGE_SIZE, to = from + PAGE_SIZE - 1,
              absDataFrag = file.contents.subseq[from,to],
              concDataFrag = findPageData[vblock,csys,d] |
              absDataFrag = concDataFrag)
      }
}

assert WriteRefinement {
  all csys, csys': ConcFsys, asys, asys': AbsFsys, d, d': Device,
      fid: FID, buffer: seq Data, offset,size : Int |
        concInvariant[csys, d] and
        writeConc[csys, csys', d, d', fid, buffer, offset, size] and
        alpha[asys, csys, d] and
        alpha[asys', csys', d']
        => writeAbs[asys, asys', fid, buffer, offset, size]
}
```

Fig. 2. Abstract Relation and Refinement Property for Write

these two distinct cases:

Phase 1: At the point of the failure, one or more pages have been programmed and their statuses have been modified to `Allocated`. To recover from this failure, we set the status of every allocated page to `Invalid`. After recovery, the device contains extra invalid pages, but to a filesystem client, the inode appears to have the same data as it did at the beginning of the operation.

Phase 2: To recover from power loss during this phase, we first invalidate every page `p1` that is paired with an allocated page `p2` (i.e. `p2` is the replacement for `p1`). Then, we validate every such `p2` by setting its status to `Valid`. In essence, the recovery process is here equivalent to completing the rest of Phase 2 that was interrupted by the power failure. At the end of the recovery, the inode contains the input data as expected by the caller of `writeConc`.

5 Analysis

Given the models for the abstract and concrete filesystems, we used the Alloy Analyzer to check refinement properties for read and write operations. First, we defined an abstraction relation `alpha` that maps a concrete state (represented by a pair of `ConcFsys` and `Device` atoms) to an abstract state (represented by an `AbsFsys` atom). The relation is expressed as a predicate (Fig. 2) that states that for every file in the abstract filesystem, the concrete state includes an inode with a correctly ordered sequence of virtual blocks containing the same data elements as in the abstract file[8].

The assertion `WriteRefinement` posits a backwards simulation for the write operation[9]. We performed backwards (rather than forwards) simulation since `alpha` maps a concrete state "upwards" to an abstract state. The predicate `concInvariant` defines a valid state in the concrete filesystem—for example, that every free page in the flash device must be completely erased—and its preservation is checked independently. When the Alloy Analyzer finds a scenario that violates the assertion within a specified scope, it graphically displays the counterexample using its built-in visualizer. In the final version of the model, the analyzer returned no counterexamples for the assertion. We used a scope of 5 for every signature in the model, with 6 flash pages, each of which was constrained to contain 4 data elements. The total size of the filesystem was therefore 24 data elements. The property was checked on a 3.6 GHz Pentium 4 machine with 3GB RAM in approximately 8 hours.

Even though the size of the filesystem that we checked is too small to represent a realistic system, we were able to find over 20 non-trivial bugs over the entire course of our design process. These bugs were removed from the model throughout the various iterations of our modeling task. In a typical filesystem, many types of errors occur in "boundary cases", which involve only a small number of components (i.e. pages, blocks, etc.). For example, consider the model for the `readConc` operation in Fig. 1. As currently shown, this operation is buggy in the following two ways: (1) If `offset` is not a multiple of `PAGE_SIZE`, then the length of the first slot in the output buffer must be less than `PAGE_SIZE`, and similarly, (2) if the expression `(offset + size)` is not a multiple of `PAGE_SIZE`, then the length of the last slot in the buffer must also be adjusted accordingly. An instance of a filesystem state with two pages is sufficient to generate a counterexample that demonstrates both of these bugs; increasing the scope to a higher value would not reveal any useful information about bugs of a similar nature.

6 Related Work

Our work is a contribution to the second pilot project in the Verified Software Repository (VSR) [3]. The idea of verifying a flash filesystem as a mini-challenge was suggested by Joshi et al. [17], and several groups are now actively working on this project [5, 7, 9, 13].

Filesystems were an early target for case studies in formal methods. As a historically significant example, Morgan and Sufrin first formalized a specification for a UNIX filesystem in Z [18]. Freitas and his colleagues refined an abstract POSIX filesystem to a concrete implementation and proved the refinement relation using Z/Eves [8]. Similarly, Arkoudas et al. proved a refinement relation between an abstract filesystem and a disk-based implementation in the Athena

[8] For simplicity, we restrict the size of every abstract file as shown in Fig. 2 to be a multiple of `PAGE_SIZE`. The complete version on the web is free of this restriction.

[9] We can obtain `ReadRefinement` by replacing the `write` predicates with `read`.

theorem prover [2]. In comparison to previous two works, which employ theorem proving, the analysis in Alloy is fully automatic, but it guarantees the correctness of the refinement relation only up to a finite bound.

Butterfield et al. formalized NAND flash memory in Z [6], following the ONFi specification, which formed a basis for our hardware model as well. Ferreira and their colleagues also formalized the ONFi specification and a POSIX filesystem in VDM++ [7]. They performed the analysis of the filesystem using the HOL theorem prover [12] and Alloy. They used the Alloy Analyzer primarily for finding a counterexample to proof obligations that could not be automatically discharged by HOL, whereas we used the analyzer to perform the analysis in its entirety.

Groce et al. performed randomized testing on a POSIX filesystem implementation that is based on NAND flash memory [13]. Yang et al. used model checking to find errors in existing filesystem implementations [25]. Although their work is not flash-specific, the nature of their analysis is similar to ours; they deliberately scaled down the size of the filesystem for increased tractability of analysis but were still able to find numerous bugs, many of which were due to complex interactions among a small number of components.

7 Discussion

In this project, we have shown that we were able use Alloy to successfully model and analyze the types of complexity that arise in a flash filesystem design. We believe that the scope we used in the final analysis was sufficient to ensure that the refinement relation is sound, but we currently cannot justify our intuition rigorously. It is also possible, as it would be even if theorem proving were used, that the concrete model harbours unintentional overconstraints that slipped through our analysis unnoticed.

Our experience has raised a number of interesting questions about the Alloy language and the analyzer. First of all, due to the declarative nature of the language, modeling multi-step operations in Alloy is not always straightforward. Although the language is expressive enough for describing such operations, writing certain types of control constructs (such as loops) in Alloy can be cumbersome, since it does not support a built-in notion of an implicit global state. In particular, in order to model changes to the device after each call to `fProgram` in `writeConc`, we explicitly introduced a sequence of state atoms and imposed a constraint between each pair of adjacent states. A language such as ASM [4]—with the notion of an implicit global state—may be more suitable for this particular aspect of the filesystem model. We are currently investigating an extension to Alloy that will provide the user with control constructs, while maintaining the declarative power of the language.

Our filesystem model, as one of the largest case studies that we have done to date, has pushed the boundary of Alloy's scalability. The analyzer uses as its backend a relational model finder called Kodkod [23] that translates an Alloy model to a CNF formula, which can then be handled by powerful SAT solvers. Although checking the refinement relation in our latest model took several hours to complete, the analyzer is fully automatic, and so we were able to leave the analyzer running unattended overnight. On the other hand, due to the exponential nature of SAT, the duration of the analysis can grow rapidly as the scope is incremented or as additional layers of complexity are added to the model. Kodkod already employs a variety of techniques to reduce the size of a SAT problem, such as symmetry breaking and sharing detection [23]; we are looking into further opportunities for an improved scalability by leveraging available techniques (e.g. additional decision procedures [11]).

Another useful feature of the Alloy Analyzer is the extraction of an unsatisfiable core [22], which highlights top-level constraints in a model that are used to establish the correctness of an assertion. In some cases, an overconstraint may cause an assertion to be vacuously true; the user can usually tell when this has happened by noticing that formulas that were expected to be highlighted as part of the core were not. The unsatisfiable core facility was very useful in this project, and did indeed expose overconstraints on several occasions. However, the granularity of the core can sometimes be too coarse to be useful to the user. In particular, a top-level formula that is existentially quantified over a conjunction of sub-constraints is treated as a single constraint in the core; this grouping might suppress potentially useful information. We are currently implementing a mechanism that will overcome this problem and extract a finer-grained unsatisfiable core.

8 Future Work

While formalizing and analyzing a design model are useful exercises on their own, one interesting question is whether the usage of the model can be extended beyond the design into the implementation and testing phases. We are looking into possible uses of the flash filesystem model. For example, by mapping our model to an existing flash filesystem implementation (such as YAFFS [1]), we can leverage the power of the Alloy Analyzer as a model finder to automatically generate a large set of test cases. Other research questions that we are planning to explore include simulation, model-based diagnosis, and code generation.

The current functionality of our filesystem is rather limited. For future work, we plan to include a larger set of POSIX file operations, such as `creat`, `open`, and `close`, and the support for directories. We also plan to model recovery mechanisms for other types of failures (besides power loss), such as bad blocks and bit corruption.

Complete versions of all Alloy models that appear in this paper are available at `http://sdg.csail.mit.edu/projects/flash`.

Acknowledgements We are grateful to Felix Chang, Greg Dennis, Vijay Ganesh, Derek Rayside, Sivan Toledo, and Emina Torlak for helpful discussions and feedback. This research was supported by the National Science Foundation under Grant Nos. 0541183 and 0438897, and by the Nokia Corporation as part of a collaboration between Nokia Research and MIT's Computer Science and Artificial Intelligence Lab.

References

1. Aleph One. YAFFS: A flash file system for embedded use. `http://www.yaffs.net`.
2. K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. On verifying a file system implementation. In: 6th ICFEM, pp. 373-390 (2004).
3. J. Bicarregui, C. A. R. Hoare, and J. Woodcock. The verified software repository: a step towards the verifying compiler. Formal Aspects of Computing, 18:143-151 (2006).
4. E. Borger and R. F. Start. Abstract State Machines: A method for high-level system design and analysis. Springer-Verlag, New York (2003).
5. M. Butler, K. Damchoom, and J-R. Abrial. Some filestore developments with Event-B and Rodin. Verifiable File Store Mini-Challenge Workshop, co-located with the 9th ICFEM (2007).
6. A. Butterfield and J. Woodcock. Formalizing flash memory: First steps. In: 12th ICECCS, pp. 251-260 (2007).
7. M. A. Ferreira, S. S. Silva, and J. N. Oliveira Verifying Intel flash file system core specification. In: 4th VDM-Overture Workshop, FM '08 (2008).
8. L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/Eves: an experiment in the verified software repository. In: 12th ICECCS, pp. 3-14 (2007).
9. L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the Verification Grand Challenge: a roadmap. In: 13th ICECCS, pp. 153, 162 (2008).
10. E. Gal and S. Toledo. Algorithms and data structures for flash memories. ACM Computing Surveys, 37:138-163 (2005).
11. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In: 19th CAV, pp.519-531 (2007).
12. M. J. C. Gordon and T. F. Melham. Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, New York (1993).
13. A. Groce, G. J. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In: 29th ICSE, pp. 621-631 (2007).
14. Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 1.0. ONFi Workgroup, `http://www.onfi.org` (2006).
15. Intel. Flash File System Core Reference Guide. Technical Report 304436001. Intel Corporation (2004).
16. D. Jackson. Software Abstractions. MIT Press, Cambridge, MA (2006).
17. R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable filesystem. In: Verified Software: Theories, Tools, Experiments (2005).
18. C. Morgan and B. Sufrin. Specification of the UNIX filing system. In: IEEE Transactions on Software Engineering, 10:128-142 (1984).
19. G. Reeves and T. Neilson. The Mars Rover Spirit FLASH Anomaly. In: IEEE Aerospace Conference (2005).
20. J. M. Spivey. The Z Notation: A Reference Manual. Prentice-Hall, NJ (1998).
21. The Open Group. The POSIX 1003.1, 2003 Edition Specification. `http://www.opengroup.org/certification/idx/posix.html`.
22. E. Torlak, F. S-H. Chang, D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In: 15th FM, pp. 326-341 (2008).
23. E. Torlak and D. Jackson. Kodkod: A relational model finder. In: 13th TACAS, pp. 632-647 (2007).
24. J. Woodcock and J. Davies. Using Z: Specification, Refinement, and Proof. Prentice-Hall, NJ (1996).
25. J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In: 6th OSDI, pp. 273-288 (2004).