Int J Software Informatics, Vol.3, No.1, January 2009, pp. 1–? E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org
 Tel: +86-10-62661040

Designing and Analyzing a Flash File System with Alloy

Eunsuk Kang and Daniel Jackson [1]

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract Alloy is a lightweight modeling language based on first-order relational logic. The language is expressive enough to describe structurally complex systems, but simple enough to be amenable to fully automated analysis. The Alloy Analyzer, with its SAT-based analysis engine, allows one to simulate traces of a system, visualize them, or search for counterexamples to a property. This article illustrates key concepts of Alloy using, as an example, the construction and analysis of a design for a flash file system. In addition to basic file operations, the design includes features that are crucial to NAND flash memory but contribute to increased complexity of the file system, such as wear leveling and erase-unit reclamation. The design also addresses the issues of fault-tolerance by providing a mechanism for recovering from unexpected hardware failures. The article describes the modeling process and discusses the results of the design analysis, which has been carried out by checking trace inclusion of the flash file system against a POSIX-compliant abstract file system.

Key words: software design; formal specification; modeling; analysis; Alloy

Eunsuk Kang and Daniel Jackson. Designing and Analyzing a Flash File System with Alloy. *Int J Software Informatics*, **2009, 1(1): 1–?. http://www.ijsi.org/1673-7288/3/1.pdf**

1 Introduction

Alloy is a lightweight language for modeling software systems. Designed with simplicity and ease of modeling in mind, the language consists of a small number of basic constructs. Yet, its underlying semantics, based on first-order relational logic, is powerful enough to express complex structures that arise in software. Alloy is *declarative*; it allows the user to model a behavior without describing its internal steps and to avoid premature implementation decisions. Alloy is also *partial*; it allows the user to focus only on those parts of the system that are deemed to be most critical.

No matter how well crafted, a model is only as useful as the questions that it seeks to answer. The Alloy Analyzer is equipped with a SAT-based engine that can generate a simulated trace of an operation, exhaustively search for a counterexample to a desired property, and present the result of analysis visually—all fully automatically.

[1] This work is sponsored 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.
Corresponding author: Eunsuk Kang, Email: eskang@mit.edu
Manuscript received ** ***. 2009; revised ** ***. 2009; accepted ** ***. 2009; published online ** ***. 2009.

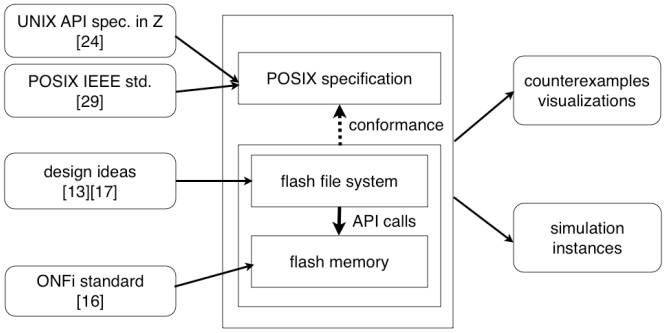

Fig. 1. Overview of the design approach

The lightweight nature of the language, combined with the capacity for automatic analysis, makes Alloy particularly suitable for *incremental* design exploration. The user may begin by writing a tiny model with one or two operations; check for design flaws by running the analysis and fix the model accordingly; and augment the design with additional features. This process may continue for several iterations until the model reaches a desired level of detail.

Alloy has been used to model and analyze a wide variety of systems, including a proton therapy machine [1], an electronic cash system [2], an architectural framework [3], cryptographic protocols [4], and semantic web ontologies [5].

This article illustrates key concepts of the language using, as an example, a recent case study in which Alloy was used to model and analyze a design of a POSIX-compliant flash file system. The Alloy model discussed in this article describes three primary aspects of a flash file system: (1) an underlying flash memory device, (2) a file system that communicates with the device to carry out file operations, and (3) a mechanism for handling hardware failures, such as unexpected power loss. A simplified version of POSIX, a popular standard for file system APIs, is used as a reference model against which the design is analyzed.

File systems have been in existence for many decades, and are generally considered to be reliable, so what is particularly challenging about designing a flash file system? Unlike disk-based storage devices, flash memory suffers from a major physical limitation; blocks can be written only a limited number of times. Overcoming this limitation requires sophisticated techniques such as wear leveling and garbage collection, which, in turn, contribute to increased complexity. This case study demonstrates how Alloy can be used to identify and analyze key properties about the design, to keep control of the complexity of the design (and hence the subsequent implementation).

Figure 1 shows an overview of the design approach for this case study. The rest of the article folllows the diagram left to right, explaining key concepts of Alloy along the way. It begins by describing a POSIX reference model (Section 2) and a model of a flash memory device (Section 3). It then presents a design of a flash file system (Section 4), which communicates with the memory and is intended to conform to the reference model. Unlike the first two models, this design does not formalize existing descriptions, but incorporates a variety of mechanisms that have appeared in the literature. Subsequent sections describe the analysis that was performed (Section 5),

discuss challenges in using Alloy (Section 6), relate this approach to others (Section 7), and list some plans for future work (Section 8).

2 POSIX File System Specification

POSIX (Portable Operating System Interface) is an international standard that specifies function signatures and behaviors of file system operations [6]. 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 file system. A specification of a UNIX file system was formalized in the Z notation [7] by Morgan et al. [8] and served as a starting point for the POSIX model in Alloy. This model is an abstract view of a file system that precludes details about an underlying hardware device. A file is modeled simply as an array of data elements, and operations as basic array manipulations.

Alloy is rooted in Z, drawing on its simple and intuitive semantics, but with a syntax and type system designed for object and data modeling. For example, a file with an array of data elements can be defined as follows[2]:

```
1 sig File {
2   contents : seq Data
3 }
4 sig Data {}
```

Then, an entire file system may be viewed simply as a container for a map that associates each file identifier with at most one file[3]:

```
1 sig AbsFsys {
2   fileMap : FID -> lone File
3 }
4 sig FID {}
```

The above four lines of Alloy constitute a complete model of a file system, albeit a static one without operations. Analysis can be immediately applied to generate an instance of the file system state, thus demonstrating the consistency of the model:

```
1 run { } for 3 but 1 AbsFsys
```

The `run` command instructs the Alloy Analyzer to find an instance with an upper bound of three elements in each signature, except for `AbsFsys`, which is limited to at most one element.

When executed, the Alloy Analyzer returns an instance that represents a file system without any files. The choice of an instance depends on the underlying SAT solver, and so is difficult to predict, but in practice, instances that are first generated tend to be small. To search for a more interesting one, the user can specify additional constraints inside the `run` command, as follows:

```
1 run { some fsys : AbsFsys | some fsys.fileMap } for 3 but 1 AbsFsys
```

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

[3] The keyword `lone` imposes a multiplicity constraint of "less than or equal to one". For example, a relation `r : A -> lone B` is a partial function from `A` to `B`.

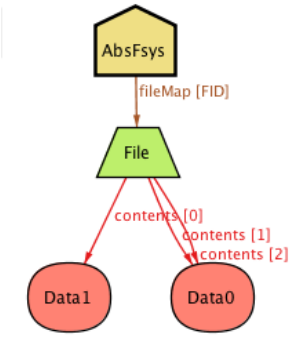

Fig. 2. An instance of the abstract file system

The constraint inside `run` states that every generated instance should include a file system with at least one entry in the file map. This time, running the command generates an instance of a file system that contains a single file with the data pattern [`Data1, Data0, Data0`]. The Alloy Analyzer can also visualize an instance in the style of object model snapshots, as shown in Figure 2[4]. The instance is small, but nevertheless illustrates the essence of the file system structure.

The basic operations of the file system—reading from and writing to a file—are modeled using functions and predicates. The read operation requires four input arguments—the current state of the file system, a file identifier, an offset, and a size. Like Z, Alloy has no built-in notion of state. A state of the file system is explicitly represented as an atom of type `AbsFsys`, and passed as an argument to operations.

The read operation first checks the input arguments to make sure that they satisfy some preconditions, which are factored into an auxiliary predicate `readPreconds`[5]. If any precondition is violated, `readAbs` returns an empty data sequence. Otherwise, it returns a sequence of the given size, starting at the offset within the file:

```
1  fun readAbs[fsys : AbsFsys, fid : FID, offset, size : Int] : seq Data {
2      readPreconds[fsys, fid, offset, size] =>
3          let file = fsys.fileMap[fid] |
4              // subseq[m,n] returns a subsequence between m and n, inclusively
5              (file.contents).subseq[offset, offset + size − 1]
6      else
7          // blanks[n] gives a sequence of blank items of length n
8          blanks[0]
9  }
10 pred readPreconds[fsys : AbsFsys, fid : FID, offset, size : Int] {
11     some fsys.fileMap[fid]  // File must exist
12     offset >= 0
13     size >= 0
14 }
```

The write operation takes a file identifier, an offset, a size, and a buffer containing

[4] Nodes represent atoms of different signatures, and edges represent relations between the atoms. Sequences are represented, as in Z, by functions from non-negative integers to elements.
[5] A list of constraints inside a predicate is implicitly conjoined.

the input data, and writes a prefix of the given size from the buffer into the file, starting at the offset[6]:

```
1  pred writeAbs[fsys, fsys' : AbsFsys, fid : FID, buffer : seq Data, offset, size : Int] {
2      writePreconds[fsys, fid, buffer, offset, size]
3      let buffer' = buffer.subseq[0, size − 1],
4          file = fsys.fileMap[fid],
5          file' = fsys'.fileMap[fid] {
6              (#buffer' = 0) =>
7                  file' = file
8              else
9                  file'.contents =
10                     (blanks[offset] ++ file.contents) ++ shift[buffer',offset]
11         promote[fsys, fsys', fid, file]
12     }
13 }
14 pred writePreconds[fsys : AbsFsys, fid : FID, buffer : seq Data, offset, size : Int] {
15    some fsys.fileMap[fid]
16    offset >= 0
17    size >= 0
18    size <= #buffer
19 }
```

Unlike `readAbs`, `writeAbs` is expressed as a predicate, rather than a function, since it may modify the state of the file system. This predicate illustrates the declarative nature of Alloy; the write operation is specified as a relation between the pre- and post-states (`fsys` and `fsys'`, respectively) of the file system, rather than a sequence of steps.

As with the read operation, the input arguments are checked to make sure that the preconditions for the write operation are satisfied. As in most Z specifications, this model conjoins preconditions with postconditions, so that when a precondition is false, the entire predicate becomes unsatisfiable, and the operation cannot occur.

After the precondition check[7], the write operation is divided into two distinct cases. First, if the input buffer is empty, `writeAbs` does not modify the contents of the file (line 7). The case in which the buffer is not empty can be further divided into two sub-cases; the offset is either located within the file, or is greater than the current file size. In the former sub-case, `writeAbs` overrides the existing data in the overlapping positions with the input data. In the latter, `writeAbs` fills the gap between the end of the file and the offset with blank data. Both of these sub-cases are handled by the single constraint on lines 9-10.

Alloy has no implicit frame conditions, so parts of the system that are not explicitly constrained by an operation are free to change. An absence of a frame condition sometimes can result in undesirable side effects. For example, the write operation should modify only the contents of a particular file and leave all other files unchanged.

[6] The cardinality operator `#` returns the length of a sequence, and `++` is the operator for relational override. The expression `shift[s,n]` represents the sequence that is like `s` but has the index of each element shifted by `n`.

[7] Here, the word "after" is used only for the narrative purpose; the order in which formulas appear inside a predicate has no semantic consequence.

```
1  sig PageAddr {} // page address
2  sig EUAddr {} // erase unit address
3
4  sig Page { data : seq Data } { #data = PAGE_SIZE }
5  sig EU { pages : PageAddr -> one Page } { #pages = ERASE_UNIT_SIZE }
6  sig Device { eraseUnits : EUAddr -> one EU } { #eraseUnits = DEVICE_SIZE }
7
8  // Reads data from the page at the given address, starting at offset
9  // to the end of the page
10 fun readPage[d : Device, pageAddr : PageAddr, euAddr : EUAddr, offset : Int] : seq Data
11 {...}
12
13 // Programs input into the page at the given address, starting at offset
14 // for the length of input, truncating input at the end of the page when necessary
15 pred programPage[d, d' : Device, pageAddr : PageAddr, euAddr : EUAddr, offset : Int,
16                  input : seq Data]
17 {...}
18
19 // Erases the entire EU at the given address
20 pred eraseEU[d, d' : Device, euAddr : EUAddr]
21 {...}
```

Fig. 3. Flash memory model

One style of specifying the scope of state changes is called *promotion* [9]. The following predicate states that the post-state of the file system is like the pre-state, except that the file entry at a particular identifier is overriden with a new file:

```
1  pred promote[fsys, fsys' : AbsFsys, fid : FID, file : File] {
2      fsys'.fileMap = fsys.fileMap ++ (fid -> file)
3  }
```

In effect, this predicate "promotes" a change in the contents of a file to a change in the entire file system, constraining all other files to remain unchanged.

3 Flash Memory

The Alloy model of flash memory is based on Open NAND Flash Interface (ONFi), an industry-wide standard for the specification of NAND flash memory [10]. It is important to note that the focus of this case study is on the design of a file system, not a flash memory device. Therefore, the model described in this section includes only the minimum amount of detail that is necessary for modeling key aspects of a flash file system.

NAND flash memory is organized into a hierarchy of components, as shown in Figure 3. The smallest unit for reading from and writing to flash memory is called a *page*. Each page consists of a fixed number of data elements[8]. The next level up in the hierarchy are *erase units* (EU), the smallest unit for erasure[9]. Finally, a *device* is the top-level component that directly communicates with the host file system.

[8] A formula `F` in `sig A {...}{F}` is a constraint that applies to every atom of type `A`.

[9] In ONFi, erase units are called blocks. To avoid naming conflations with file blocks (each of which maps to a page rather than a block in flash), ONFi blocks have been renamed to erase units in the Alloy model.

During a file operation, the host file system may make one or more calls to three flash API functions: read, program, and erase. Due to limited space, this section presents only the interface declarations of these operations. Note that `programPage` and `eraseEU` are expressed as predicates, since these operations may modify the state of the device.

Several characteristics of flash memory contribute to the complexity of the file system design. Unlike sectors in disks, pages in flash must be erased before they can be re-programmed with a new data pattern. The mismatch between the granularity of the erasure and programming operations makes it difficult to selectively rewrite parts of the memory. In addition, each EU can be erased a limited number of times, after which the unit becomes unreliable. Thus, when erase operations are carried out without careful considerations to this limitation, a flash device quickly becomes unusable. A technique, called *wear leveling*, plays an essential role in prolonging the lifetime of flash memory, and is described in detail later in this article.

4 Flash File System

This section describes an Alloy model of a flash file system that communicates with the flash device to perform file operations. This model is not based on a particular existing flash file system. Rather, it incorporates a variety of mechanisms that have appeared in literature: wear-leveling and erase-unit reclamation from Gal and Toledo's survey paper on flash memory algorithms [11], and a mechanism for power-loss recovery from the Intel Flash File System Specification [12]. It is important to note that the primary concern of this modeling task is the correctness of the design, not its performance. Thus, when multiple techniques were available, the simplest alternative was chosen.

Figure 4 shows the overall structure of the flash file system state in Alloy. A file, represented by an `Inode`, consists of a list of blocks, each of which maps to a page on the flash device. Like its abstract counterpart, `ConcFsys` maps each file identifier to at most one inode. The file system interacts with the flash device through an intermediate layer called `blockManager`, which maintains an one-to-one mapping between all blocks and flash pages, as well as a list of currently free blocks. The block manager also keeps a garbage collector, which reclaims obsolete blocks on-demand (described in Section 4.1.2).

4.1 Concrete Operations

The two basic file operations described here—read and write—are substantially more complex than their counterparts in the abstract file system. Rather than consisting of simple array manipulations, a concrete operation may involve multiple invocations of one or more flash API functions, since each inode consists of a number of fixed-size blocks. Due to limited space, only the most distinctive aspects of the operation are described in this article.

4.1.1 Concrete Read Operation

Like `readAbs`, `readConc` is a function that takes four arguments—`fsys`, `fid`, `offset`, and `size`—and returns a sequence of data[10]:

[10] The preconditions for `readConc` are essentially the same as those for `readAbs`.

```
sig Inode {
  blockList : seq Block,
  eofIdx : Int // index in the last block indicating EOF
}{
  eofIdx >= 0
  eofIdx < BLOCK_SIZE
  not blockList.hasDups // Inode cannot have duplicate blocks
}
sig Block {}
sig ConcFsys {
  inodeMap : FID -> lone Inode,
  blockManager : BlockManager
}
sig BlockManager {
  blockMap : Block one -> one PhysicalAddr,
  freeBlockList : seq Block,
  device : Device,
  garbageCollector : GarbageCollector
}{
  not freeBlockList.hasDups
}
sig GarbageCollector {
  obsoleteBlocks : set Block,
  reserveEU : EUAddr,
  eraseFreq : EUAddr -> one EraseFreq
}
// Each physical address points to a page on the flash memory
sig PhysicalAddr {
  euAddr : EUAddr,
  pageAddr : PageAddr
}
```

Fig. 4. Flash file system structure

```
fun readConc [fsys : ConcFsys, fid : FID, offset, size : Int] : seq Data {
    readPreconds[fsys, fid, offset, size] =>
        readInode[fsys, fid, offset, size]
    else
        blanks[0]
}
```

The core of the read operation, specified in a helper function (`readInode`), involves reading each of blocks in the inode and concatenating all of the data into a single output sequence. This operation is modeled with a set comprehension:

```
fun readInode[fsys : ConcFsys, fid : FID, offset, size : Int] : seq Data {
    let bm = fsys.blockManager,
        inode = fsys.inodeMap[fid],
        eofIdx = inode.eofIdx |
        {i : Int, d : Data |
            let blockIndex = (i + offset) / BLOCK_SIZE,
                dataIndex = (i + offset) % BLOCK_SIZE,
                blockData = readBlock[bm, 0, inode.blockList[blockIndex]] |
                    i >= 0 and i < size and
```

```
10                  ((blockIndex = inode.blockList.lastIdx) =>
11                      dataIndex <= eofIdx) and
12                  blockData[dataIndex] = d
13          }
14  }
```

The declaration of the set comprehension (line 5) indicates that elements in the set are of type `Int -> Data` (which is the type of `seq Data` in Alloy). The body of the comprehension follows, beginning with a series of `let` bindings (lines 6-8). The variable `blockIndex` holds the index of the current inode block to be read. The variable `dataIndex` holds the index of the data element that corresponds to i^{th} element in the final output sequence. The variable `blockData` stores the result from `readBlock`, which, in turn, invokes `readPage` on the flash page that corresponds to the inode block.

A series of constraints follows the let bindings (lines 9-12). The constraint on line 9 bounds the length of the output sequence by `size`. As a special case, if the block currently being read is the last one in the inode, then data elements beyond the end of file index (`eofIdx`) must be excluded (lines 10-11). Finally, the last constraint ensures that each slot in the output sequence stores the corresponding data element from the block (line 12).

4.1.2 Wear-Leveling and Garbage Collection

Wear-leveling is a technique for distributing erasures across flash memory. To overcome the inherent limitations of the hardware, most flash file systems perform wear-leveling incrementally as an integral part of the operations.

The following example illustrates the standard wear-leveling technique [11]. Suppose an inode consists of a list of blocks, one of which (`blk`) is mapped to a flash page `p1`. A client sends a request to the file system to overwrite the existing data, including `blk`, in the inode. A simple approach would be to erase the EU that contains `p1` and then program new data into `p1`. However, if operations involving the inode were frequent, then this EU would wear out much more quickly than others. Thus, rather than erasing `p1`, the file system writes the new data into a free, available page `p2`. In addition, it marks the data in `p1` as obsolete and updates `blockMap` in `ConcFsys` to indicate that `blk` is now mapped to `p2`.

Over time, the flash device accumulates obsolete data and eventually runs out of available blocks. In order to free up space for new program operations, the file system executes garbage collection in the following steps:

1. Search for all EUs that contain obsolete data. One of these EUs is selected for erasure; the selection policy is a decision left to the designer of the system. The authors' design chooses the EU that has been least often erased by checking `eraseFreq` (line 25, Figure 4) in `GarbageCollector`.
2. Relocate all valid pages in the selected EU. This EU (call it `dirtyEU`) may still contain one or more pages that hold valid data. For the purpose of relocation, the file system keeps one completely erased EU as a spare (`reserveEU` on line 24, Figure 4). Each valid page is relocated from `dirtyEU` to the corresponding position in `reserveEU`.

3. Erase `dirtyEU` using the `eraseEU` command. This EU becomes the new reserve EU for the file system.

Now, all pages within the old `reserveEU` that do not hold the relocated data from `dirtyEU` are free and available for programming.

4.1.3 Concrete Write Operation

The concrete write operation consist of three major steps: (1) checking preconditions, (2) writing data into blocks, and (3) updating relevant metadata about the inode and blocks:

```
1  pred writeConc [fsys, fsys' : ConcFsys, fid : FID, buffer : seq Data, offset, size : Int] {
2      writePreconds[fsys, fid, buffer, offset, size]
3      writeBlocks[fsys, fsys', fid, buffer, offset, size]
4      updateMetadata[fsys, fsys', fid, offset, size]
5  }
```

The core of the operation, specified in `writeBlocks`, involves partitioning the input buffer into fixed-size fragments and programming them into the flash memory one-by-one:

```
1  pred writeBlocks[fsys, fsys' : ConcFsys, fid : FID, buffer : seq Data, offset, size : Int] {
2      let relativeOffset = offset % BLOCK_SIZE,
3          bufferHeadSize = smaller[BLOCK_SIZE − relativeOffset, size],
4          bufferHead = buffer.subseq[0, bufferHeadSize − 1],
5          bufferTail = buffer.subseq[bufferHeadSize, size − 1],
6          writeBlockList = computeWriteBlockList[fsys, fid, offset, size] |
7              writeFirstBlock[fsys, fsys', relativeOffset,
8                              writeBlockList.first, bufferHead] and
9              writeOtherBlocks[fsys, fsys', writeBlockList.rest, bufferTail]
10 }
```

To simplify the operation, the input buffer is first divided into two parts—the head (line 4) and the tail (line 5). The head corresponds to the buffer prefix whose length may be less than the size of a block, depending on `offset` and `size`. On line 6, a helper predicate, `computeWriteBlockList`, carries out a series of calculations, based on `offset` and `size`, to derive a list of blocks to be written. This list may include blocks that currently belong to the inode as well as new ones that will be appended to the inode, if any.

The two predicates that follow on lines 7 and 9, `writeFirstBlock` and `writeOtherBlocks`, carry out the actual process of data modification. The first writes the head of the buffer into the first block in `writeBlockList`. The second writes the tail of the buffer into the remaining blocks:

```
1  pred writeOtherBlocks[fsys, fsys' : ConcFsys, block : seq Block, buffer : seq Data] {
2      let bm = fsys.blockManager, bm' = fsys'.blockManager |
3          all idblocks.inds |
4              let block = blocks[idx],
5                  fromIdx = BLOCK_SIZE∗idx,
6                  toIdx = fromIdx + BLOCK_SIZE − 1,
7                  bufferFragment = buffer.subseq[fromIdx, toIdx] |
8                      writeBlock[bm, bm', 0, block, bufferFragment]
```

```
9 }
```

The operation carried out in this predicate resembles a loop. For each `idx`, which corresponds to the index of the current block to be written, `writeOtherBlocks` extracts a data fragment of length `BLOCK_SIZE` from the buffer (lines 4-7). Then, on line 8, a helper predicate, `writeBlock`, programs the data fragment into a flash page by invoking `programPage`.

After writing all of the buffer into the blocks, `writeConc` updates the relevant metadata in `updateMetadata`. If the operation involves writing data beyond the current end of the file, then one or more blocks may be removed from `freeBlockList` and added to `blockList` in `Inode`. For each existing block that is overwritten with new data, `updateMetadata` updates `blockMap` with a new block-to-page mapping. In addition, `updateMetadata` includes frame conditions to make sure only the information about the relevant inode and blocks changes, and all other parts of the system remain the same.

It is important to note that the authors' operational tone in the description of `writeConc` is intentionally informal. Alloy has no built-in operational semantics, and the write operation is modeled in a purely declarative style.

4.2 Fault Tolerance

Over the course of its lifetime, a flash device is susceptible to a variety of hardware failures. One crucial aspect of designing a flash file system is robustness in recovering from such failures. After recovery, the file system must be either in a state as if an operation has never begun, or in a state where the operation has been successfully completed. This section illustrates one particular type of fault-tolerance mechanism—recovery from power loss in the middle of a write operation.

Assuming that programming a page is atomic[11], how should the designer ensure that a file write, which consists of a series of program operations, appear atomic as well? For this purpose, the file system keeps extra metadata about flash pages. Each page is associated with one of four status constants:

```
1 abstract sig PageStatus {}
2 one sig Free, // Erased and ready to be programmed
3         Allocated, // Allocated for a write operation
4         Valid, // Valid for a read operation
5         Invalid extends PageStatus {} // Invalid data, not to be read
```

Modification of page status occurs over two phases during a write operation. In Phase 1 (inside `writeBlocks`), the file system programs free pages with input data and modifies their status to `Allocated`. In Phase 2 (inside `updateMetadata`), the file system invalidates the pages that contain obsolete data and validates those that were allocated during Phase 1. A power failure occurs during one of the two phases, and on reboot, the file system examines the status of the pages to carry out the recovery process.

Crash during Phase 1: At the point of the failure, one or more pages have been

[11] In general, failures may occur during low-level flash operations, and can be detected by examining the status register in the flash device. The version of the Alloy model discussed in this article does not model this aspect of the hardware.

programmed and their statuses have been set to `Allocated`. To recover from this failure, the file system marks every such allocated page as `Invalid`. After recovery, the device contains extra invalid pages, but to a file system client, the inode appears to have the same data as it did at the beginning of the operation. These invalid pages can subsequently be freed by the garbage collector.

Crash during Phase 2: For every page `p1` that has been replaced by an allocated page `p2`, the file system validates `p2` and invalidates `p1`. In essence, the recovery process here is equivalent to completing the rest of Phase 2 that was interrupted by the power failure. After recovery, the inode contains the input data as expected by the caller of `writeConc`.

5 Analysis

This section describes different types of analysis that were carried out on the model of the flash file system.

5.1 Properties of the Design

Having sketched out an initial model of the flash file system, the designer may wish to gain confidence about the design through rigorous validation. In Alloy, an *assertion* is a constraint that is intended to hold in all cases. For instance, one desirable property of the concrete write operation may be phrased as follows:

```
1 assert WriteOK {
2     all fsys, fsys' : ConcFsys, fid : FID, offset, size : Int, buffer : seq Data |
3         writeConc[fsys, fsys', fid, buffer, offset, size]
4     =>
5         readConc[fsys', fid, offset, size] = buffer.subseq[0, size − 1]
6 }
```

This assertion states that after each successful completion of the write operation, the data that is immediately read from the file system is equal to the data in the input buffer[12]. A `check` command, given a bound on the number of elements in each signature, analyzes every possible instance of the operation up to the bound. In particular, the following instance of `check` places a bound of 5 on every signature, except `ConcFsys`, which is limited to only two atoms—sufficient to represent pre- and post-states of the file system:

```
1 check WriteOK for 5 but 2 ConcFsys
```

The analysis is *exhaustive*; if a counterexample exists within the bound, the Alloy Analyzer is guaranteed to find it. However, the analysis is also *bounded*; the successful run of `check` does not necessarily imply that the assertion is valid, since a counterexample may exist in a higher scope.

Is the above assertion sufficient to guarantee that the write operation is free of errors? Not necessarily. Like in other formal specification methods, models in Alloy runs into the risk of underspecification, and a design will be only as "correct" as the user exercises it to be. For example, how should the designer ensure that the write

[12] This assertion, in turn, depends on the correctness of `readConc`, which can be checked through simulation or another assertion.

operation does not inadvertently program pages that have not been erased yet? An additional assertion, similar to the following one, may be necessary:

```
1  assert WearLevelingOK {
2      all fsys, fsys' : ConcFsys, fid : FID, offset, size : Int, buffer : seq Data |
3          writeConc[fsys, fsys', fid, buffer, offset, size]
4      =>
5          all addr : PhysicalAddr |
6              let dev = fsys.blockManager.device,
7                  dev' = fsys'.blockManager.device,
8                  prePageData = readPage[dev, addr.pageAddr, addr.euAddr, 0],
9                  postPageData = readPage[dev', addr.pageAddr, addr.euAddr, 0] |
10                     prePageData.elems not in ErasedData =>
11                     prePageData = postPageData
12 }
```

This assertion states that during every instance of the write operation, the content of every page that had not been entirely erased before the operaton must remain unchanged. In an early version of the model, the authors ran both `WriteOK` and `WearLevelingOK`. The analyzer returned no counterexample for the former, but found a flaw in which an incorrect frame condition would allow a non-erased page to change its content during the operation.

The final version of the Alloy models, available online[13], includes other assertions about the file operations, the flash device, and the fault-tolerance mechanism, many of which are not discussed here due to limited space.

5.2 Consistency of the Design

The flexibility of declarative languages, like Alloy or Z, comes with a risk. Given an unrestricted usage of conjunctions and negations, the designer may inadvertently include logical statements that overconstrain the model. Properties will hold vacuously true, but the user will enjoy a false sense of accomplishment in having designed a flawless system, only to discover later that the design is, in fact, not implementable.

Simulation, introduced earlier in Section 2, can be used as a safeguard against such inconsistency. If the analyzer fails to generate an instance that satisfies an expected predicate, then one or more over-constraints may exist in the model. For example, after completing the model of the flash file system, the designer may wish to generate and examine sample states of the file system using a `run` command:

```
1  pred show[fsys : ConcFsys] {
2      some fid : FID | some fsys.inodeMap[fid].blockList
3  }
4  run show for 3
```

When executed, the analyzer fails to generate an instance. The result may be surprising; `show` simply states that the generated state should have at least one non-empty inode. This suggests that one or more overconstraints are lurking in the model, but looking for them is not a trivial task, especially if the size of the model is large.

One useful feature of the Alloy Analyzer is the extraction of an unsatisfiable

[13] The URL for the complete Alloy models is: `http://sdg.csail.mit.edu/projects/flash`

core [13], which corresponds to a set of constraints that contribute to the unsatisfiability (and therefore, lack of an instance) of a formula. When the analyzer fails to generate a simulation instance, it displays the core to the user, suggesting parts of the model that should be examined for overconstraints. For example, for the above unsatisfiable predicate `show`, the analyzer highlights the following fact in the model [14]:

```
1 fact InodeBlocksAreDisjoint {
2     all i1, i2 : fsys.inodeMap[FID] |
3         no i1.blockList.elems & i2.blockList.elems
4 }
```

This constraint expresses an invariant that two inodes do not share blocks. A close look at the formula reveals that it actually says more than intended; it does not allow an inode to share blocks with itself, implying that every inode must be empty! One remedy is to restrict the bindings of `i1` and `i2` to be distinct using the keyword `disj`:

```
1 fact InodeBlocksAreDisjoint {
2     all disj i1, i2 : fsys.inodeMap[FID] |
3         no i1.blockList.elems & i2.blockList.elems
4 }
```

Unsatisfiable cores are also useful in confirming the coverage of an assertion. Sometimes, the analyzer, after failing to find a counterexample to a property, returns a core that highlights only a small portion of the model. Two possibilities arise: (1) the assertion is too weak, indicating that the property may not be saying as much as the designer thinks it does, or (2) some parts of the system are irrelevant to the property, hinting that they are useless components that could be safely eliminated. Neither of the two cases is as harmful as an insidious overconstraint, but they nevertheless warrant repair or re-thinking of the design.

5.3 Conformance against POSIX

Having achieved confidence that the design of the flash file system is indeed a functional and viable one, how should the user ensure that this design conforms to the POSIX specification? This section describes one particular notion of conformance called *trace inclusion*—namely that, the set of behaviors of a system (the flash file system) is a subset of the behaviors of another (the POSIX specification).

In Figure 5, `WriteConformance` states an inductive definition for the trace inclusion of the write operation. The assertion says that every time the flash file system takes a step through `writeConc`, the specification must be able to match that step through `writeAbs`. Trace inclusion allows a simple and intuitive formulation of counterexamples; a violation of the specification is any concrete transition for which there is no valid transition in the abstract system[15].

[14] A fact is a constraint that is assumed to always hold.

[15] Trace inclusion alone would consider a concrete system that does nothing useful to be conformant to the abstract system. The designer may wish to check an additional property that requires the concrete system to match every abstract operation. However, a formulation of such a property generally includes an *unbounded universal quantifier*, which is undesirable due to the bounded nature of the analysis in Alloy. This problem is discussed in more detail in Section 5.3 of [14].

```
1  pred alpha [concFsys : ConcFsys, absFsys : AbsFsys] {
2      all fid : FID {
3      let inode = concFsys.inodeMap[fid], file = absFsys.fileMap[fid],
4          blocks = inode.blockList, eofIdx = inode.eofIdx |
5              #file.contents = (#blocks − 1)∗BLOCK_SIZE + eofIdx + 1 and
6              (all idblocks.inds |
7               let block = blocks[idx],
8                   dataFrag = findPage[concFsys, block].data,
9                   from = BLOCK_SIZE∗idx,
10                  to = from + BLOCK_SIZE − 1 |
11                      (idx = blocks.lastIdx) =>
12                          file.contents.subseqFrom[from] =
13                              dataFrag.subseq[0, eofIdx]
14                      else
15                          file.contents.subseq[from, to] = dataFrag
16             )
17      }
18 }
19
20 assert WriteConformance {
21     all concFsys, concFsys' : ConcFsys, absFsys, absFsys' : AbsFsys,
22         fid : FID, buffer : seq Data, offset, size : Int |
23             concInvariants[concFsys] and
24             writeConc[concFsys, concFsys', fid, buffer, offset, size] and
25             alpha[concFsys, absFsys] and
26             alpha[concFsys', absFsys']
27         =>
28             writeAbs[absFsys, absFsys', fid, buffer, offset, size]
29 }
```

Fig. 5. Abstraction relation and trace inclusion property for the write operation

The flash file system model contains much more information than the POSIX specification, so only the parts of a concrete state that are externally visible (i.e. contents of an inode) are compared to an abstract state. An abstraction relation, expressed in the predicate `alpha`, maps concrete states to corresponding abstract states[16]. The relation says that for every file in the abstract file system, each related concrete file system includes an inode with a correctly ordered sequence of blocks containing the same data elements as in the abstract file.

Only the set of behaviors that originate from a valid concrete state needs to be checked for conformance. The predicate `concInvariants` (line 23) defines the set of reachable states in the flash file system—for example, that every free page in the flash device must be completely erased—and its preservation is checked independently.

The analysis is performed incrementally, starting at a tiny file system, and gradually increasing the scope to a reasonable size. In the final version of the model, the analyzer returned no counterexamples for the assertion. The largest analyzable scope was 5 for every signature in the model, with 6 flash pages, each of which was con-

[16] In a traditional formulation of refinement, `alpha[concFsys', absFsys']` would appear on the consequent of the implication, introducing an unbounded universal quantifier when the assertion is negated. The formulation shown in Figure 5 is sound, given that `alpha` is both functional and total.

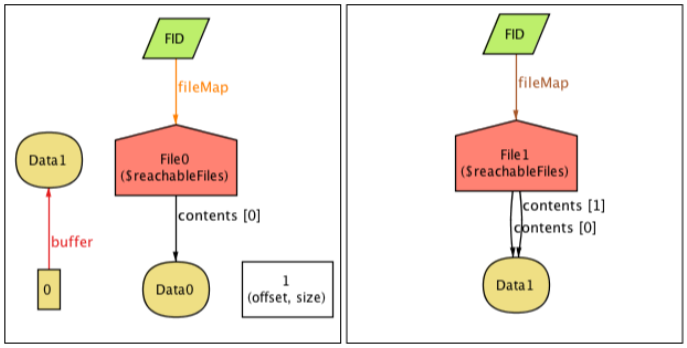

Fig. 6. A counterexample: Pre- and post-states of the abstract file system

strained to contain 4 data elements. The total size of the file system 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 file system that was analyzed is too small to represent a realistic system, over 20 non-trivial bugs were discovered as design modeling progressed. These bugs were removed from the model throughout the various iterations of the modeling and analysis task. The effectiveness of the bounded analysis relies upon an intuition (called "small-scope hypothesis" [14]) that many types of errors can be revealed through only a small number of components.

Figure 6 shows one of the counterexamples that were generated during the analysis. The visualization depicts the states of the abstract file system before and after an operation that involved writing a single-element buffer [`Data1`] into a file starting at an offset of 1. The content of the file changed from [`Data0`] to [`Data1, Data1`] (instead of the expected value [`Data0, Data1`]). This result is clearly not allowed by `writeAbs`. So how did `writeConc` produce this behavior? A close inspection of the Alloy model reveals a design flaw in `writeConc`; when overwriting an existing block at an offset that is greater than 0, the data fragment prior to the offset must be preserved in the replacement page. An instance of a file system with a single-block file is sufficient to generate this counterexample, and increasing the scope would not reveal any additional information.

6 Discussion

This case study has demonstrated how one may use Alloy to tackle the types of complexity that arise in a flash file system. The first author of the project had little prior knowledge in either Alloy or flash file systems, but was able to model and analyze a prototype design with an interesting set of features in approximately 3 months. Challenges were encountered during the process, however, and many of them raise interesting questions about Alloy and its analyzer.

Analysis Scope The analysis is exhaustive but bounded, and therefore, the analyzer can provide only a certain degree of confidence, not a guarantee, that the system satisfies a property. The authors believe that the scope in the final analysis (a file system with a maximum of 24 flash pages) was sufficient to ensure that the design is correct, but they currently cannot justify this intuition rigorously. It is also possible,

as it would be even if theorem proving were used, that the Alloy models harbor unintentional overconstraints that slipped through the analysis unnoticed.

Modeling style The flexibility of Alloy, stemming from its declarative nature as well as the absence of a hardwired idiom, is not without its costs. The explicit representation of states, using distinct signature atoms (e.g. `fsys` and `fsys'`), can become cumbersome for writing multi-step operations. Consider simulating a trace that consists of, in order, a file write, garbage collection, and another write. A sequence of states must be introduced using quantifiers, and a constraint must be imposed between each pair of adjacent states in the sequence. This is a minor issue, but an annoyance nevertheless.

The lack of implicit frame conditions is a mixed blessing. Promotion works nicely for `writeAbs`, but writing one for `writeConc`, where multiple, separate components of the system changes, is less straightforward. A frame condition is needed at the flash device level to constrain pages that are not modified by the write; another at the block manager level to frame the changes in `blockMap` and `freeBlockList`; and yet another to ensure the metadata for only a particular inode changes. Framing is a well-known problem with a number of proposed solutions [15], but it still remains a nuisance that directs the user's attention and effort away from the design task.

A language such as ASM [16] or SPEC [17]—with the notion of an implicit global state—may be more suitable for modeling dynamic aspects of a system. A long-standing research challenge remains to be solved: developing an extension to Alloy that will provide the user with control constructs, while maintaining the declarative power of the language.

Scalability The flash file system model, as one of the largest case studies in Alloy, pushes the boundary of the analyzer's scalability. The tool uses as its backend a relational model finder called Kodkod [18] that translates an Alloy model to a CNF formula, which can then be handled by powerful SAT solvers. Although checking the trace inclusion property in the latest model took several hours to complete, the analyzer is fully automatic, and so the analyzer can be left running unattended overnight. On the other hand, due to the exponential nature of SAT, the duration of the analysis grows 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 the SAT problem, such as symmetry breaking and sharing detection [18], but there are no doubt opportunities for further advances.

Unsatisfiable Core As mentioned earlier, the unsatisfiable core facility was very useful in this project, and exposed 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. The authors are currently implementing a mechanism that will overcome this problem and extract a smaller core.

7 Related Work

This work is a contribution to the second pilot project in the Verified Software Repository (VSR) [19]. The idea of verifying a flash file system as a mini-challenge was suggested by Joshi et al. [20], and several groups have also contributed to this project [21, 22, 23, 24, 25, 26]. This article expands on a preliminary version of the authors' work [27], which was presented at the VSR-net workshop [28].

File systems were an early target for case studies in formal methods. Morgan and Sufrin first formalized a specification for a UNIX file system in Z [8]. Freitas and his colleagues refined an abstract POSIX file system to a concrete implementation and proved the refinement relation using Z/Eves [29]. Similarly, Arkoudas et al. proved a refinement relation between an abstract file system and a disk-based implementation in the Athena theorem prover [30]. In comparison to the previous two works, which employ theorem proving, the analysis in Alloy is fully automatic, but it guarantees the correctness of the design only up to a finite bound.

Butterfield et al. formalized NAND flash memory in Z [22], following the ONFi specification, which formed a basis for the authors' flash model as well. Ferreira and their colleagues also formalized the ONFi specification and a POSIX file system in VDM++ [24]. They performed the analysis of the file system using the HOL theorem prover [31] and Alloy. They used the Alloy Analyzer primarily for finding a counterexample to proof obligations that could not be automatically discharged by HOL, whereas the authors used the analyzer to perform the analysis in its entirety.

Groce et al. performed randomized testing on a POSIX file system implementation that is based on NAND flash memory [26]. Kim et al. used a SAT-based model checker to analyze a commercial flash file system [32]. Similarly, Yang et al. used model checking to find bugs in existing file system implementations [33]. The nature of the previous two groups' analysis is similar to the authors'; they deliberately scaled down the size of the file system 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.

8 Future Work

While formalizing and analyzing a design model are useful exercises on their own, one interesting question is whether the model can be exploited beyond the design into the implementation and testing phases. The authors are looking into possible uses of the flash file system model. For example, by mapping the Alloy model to an existing flash file system implementation (such as YAFFS [34]), one might leverage the power of the Alloy Analyzer as a model finder to automatically generate a large set of test cases. Other research questions that the authors are planning to explore include simulation, model-based diagnosis, and code generation.

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

Acknowledgements The authors 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] R. Seater and D. Jackson. Requirement progression in problem frames applied to a proton therapy system. 14th RE, pp. 166-175 (2006).

[2] T. Ramananandro. Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method. Formal Aspects of Computing, 20(1):21-39 (2008).

[3] D. Jackson and K. Sullivan. COM revisited: tool-assisted modelling of an architectural framework. 8th FSE, 149-158 (2000).

[4] E. Torlak, M. v. Dijk, B. Gassend, D. Jackson, and S. Devadas. Knowledge flow analysis for security properties. CoRR abs/cs/0605109 (2006).

[5] J. S. Dong, J. Sun, H. Wang. Checking and reasoning about semantic web through Alloy. 12th FME, pp. 796-813 (2003).

[6] The Open Group. The POSIX 1003.1, 2003 Edition Specification. `http://www.opengroup.org/certification/idx/posix.html`.

[7] J. M. Spivey. The Z Notation: A Reference Manual. Prentice-Hall, NJ (1998).

[8] C. Morgan and B. Sufrin. Specification of the UNIX filing system. IEEE Transactions on Software Engineering, 10:128-142 (1984).

[9] J. Woodcock and J. Davies. Using Z: Specification, Refinement, and Proof. Prentice-Hall, NJ (1996).

[10] Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 1.0. ONFi Workgroup, `http://www.onfi.org` (2006).

[11] E. Gal and S. Toledo. Algorithms and data structures for flash memories. ACM Computing Surveys, 37:138-163 (2005).

[12] Intel. Flash File System Core Reference Guide. Technical Report 304436001. Intel Corporation (2004).

[13] E. Torlak, F. S-H. Chang, D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. 15th FM, pp. 326-341 (2008).

[14] D. Jackson. Software Abstractions. MIT Press, Cambridge, MA (2006).

[15] A. Bordiga, J. Mylopoulos, R. Reiter. On the Frame Problem in Procedure Specifications. IEEE Transactions on Software Engineering, 21(10):785-798 (1995).

[16] E. Borger and R. F. Start. Abstract State Machines: A method for high-level system design and analysis. Springer-Verlag, New York (2003).

[17] B. Lampson. 6.826 course notes. http://web.mit.edu/6.826/www/notes (retrieved on Jan. 31, 2009).

[18] E. Torlak and D. Jackson. Kodkod: A relational model finder. 13th TACAS, pp. 632-647 (2007).

[19] J. Bicarregui, C. A. R. Hoare, and J. Woodcock. The verified software repository: a step towards the verifying compiler. Formal Aspects of Computing, 18(2):143-151 (2006).

[20] R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable filesystem. Verified Software: Theories, Tools, Experiments (2005).

[21] K. Damchoom, M. Butler, and J-R. Abrial. Modelling and proof of a tree-structured file system in Event-B and Rodin. 10th ICFEM, pp.25-44 (2008).

[22] A. Butterfield, L. Freitas and J. Woodcock. Mechanising a formal model of flash memory. Science of Computer Programming, 74(4):219-237 (2008).

[23] F. Dadeau, A. D. Kermadec, and R. Tissot. Combining scenario- and model-based testing to ensure POSIX compliance. 1st ABZ, pp. 153-166 (2008).

[24] M. A. Ferreira, S. S. Silva, and J. N. Oliveira Verifying Intel flash file system core specification. 4th VDM-Overture Workshop, FM '08 (2008).

[25] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the Verification Grand Challenge: a roadmap. 13th ICECCS, pp. 153, 162 (2008).

[26] A. Groce, G. J. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. 29th ICSE, pp. 621-631 (2007).

[27] E. Kang and D. Jackson. Formal modeling and analysis of a flash filesystem in Alloy. 1st ABZ, pp. 294-308 (2008).

[28] J. Woodcock and P. Boca. ABZ2008 VSR-net workshop. 1st ABZ, pp. 378-379 (2008).

[29] L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/Eves: an experiment in the verified software repository. 12th ICECCS, pp. 3-14 (2007).

[30] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. On verifying a file system implementation. 6th ICFEM, pp. 373-390 (2004).

[31] 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).

[32] M. Kim, Y. Kim, and H. Kim. Unit Testing of Flash Memory Device Driver through a SAT-Based Model Checker. 23rd ASE, pp. 198-207 (2008).

[33] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. 6th OSDI, pp. 273-288 (2004).

[34] Aleph One. YAFFS: A flash file system for embedded use. `http://www.yaffs.net`.