

Formal Modeling and Analysis of a Flash File System in Alloy

Eunsuk Kang & Daniel Jackson

MIT



ABZ 2008 September 17, London, UK

Flash memory

- ▶ Increasingly popular as storage device

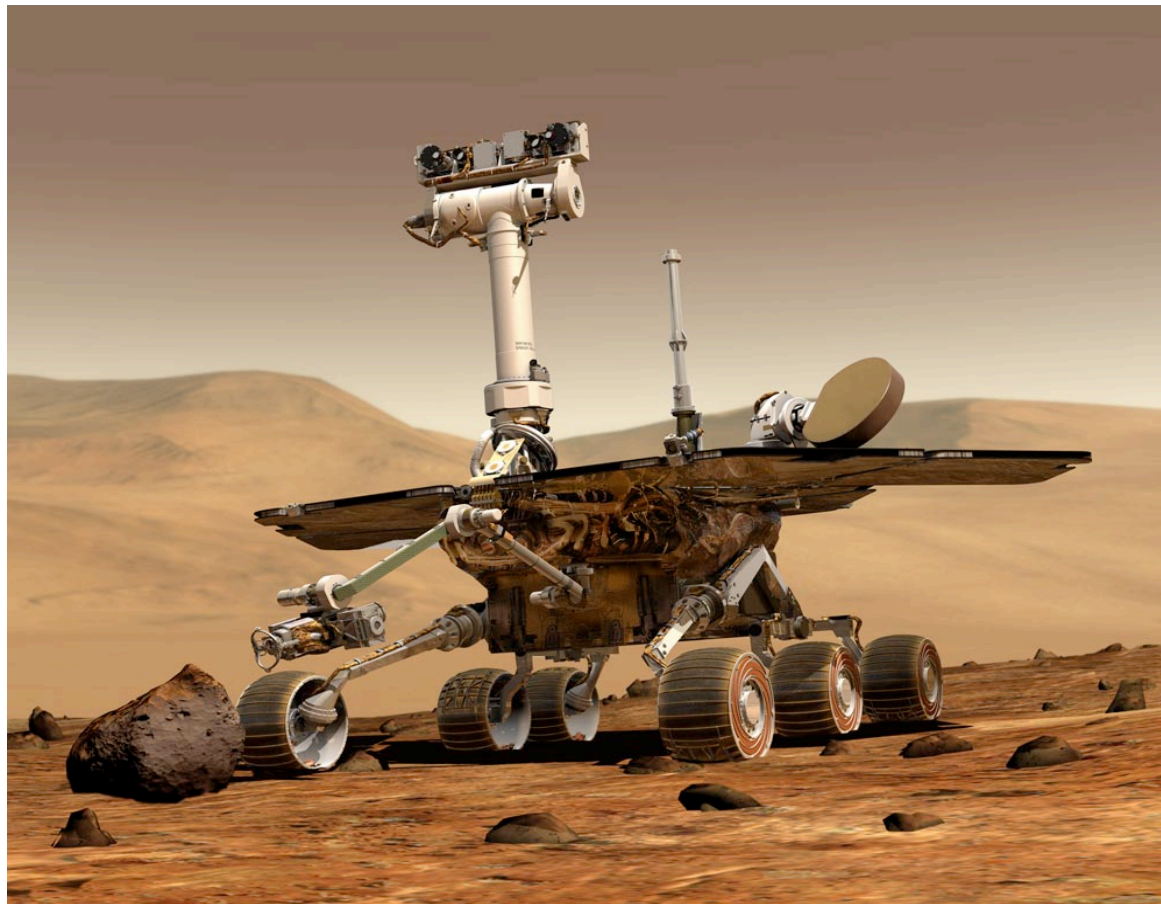
- ▶ **Benefits:**

 - High durability, low power consumption

- ▶ **Challenges:**

 - Limited lifetime (requires sophisticated techniques),
high **complexity**, low **reliability**

NASA Mars Exploration Rover *Spirit*



On-board flash memory to store scientific data

Flash anomaly on *Spirit*

- ▶ System failure 18 days after landing (2004)
- ▶ **Cause:** Design flaw in the flash file system
- ▶ “There was a belief among the FSW development team that the system would not exhibit the behavior that is the root cause of the anomaly...” [Reeves, 2004]

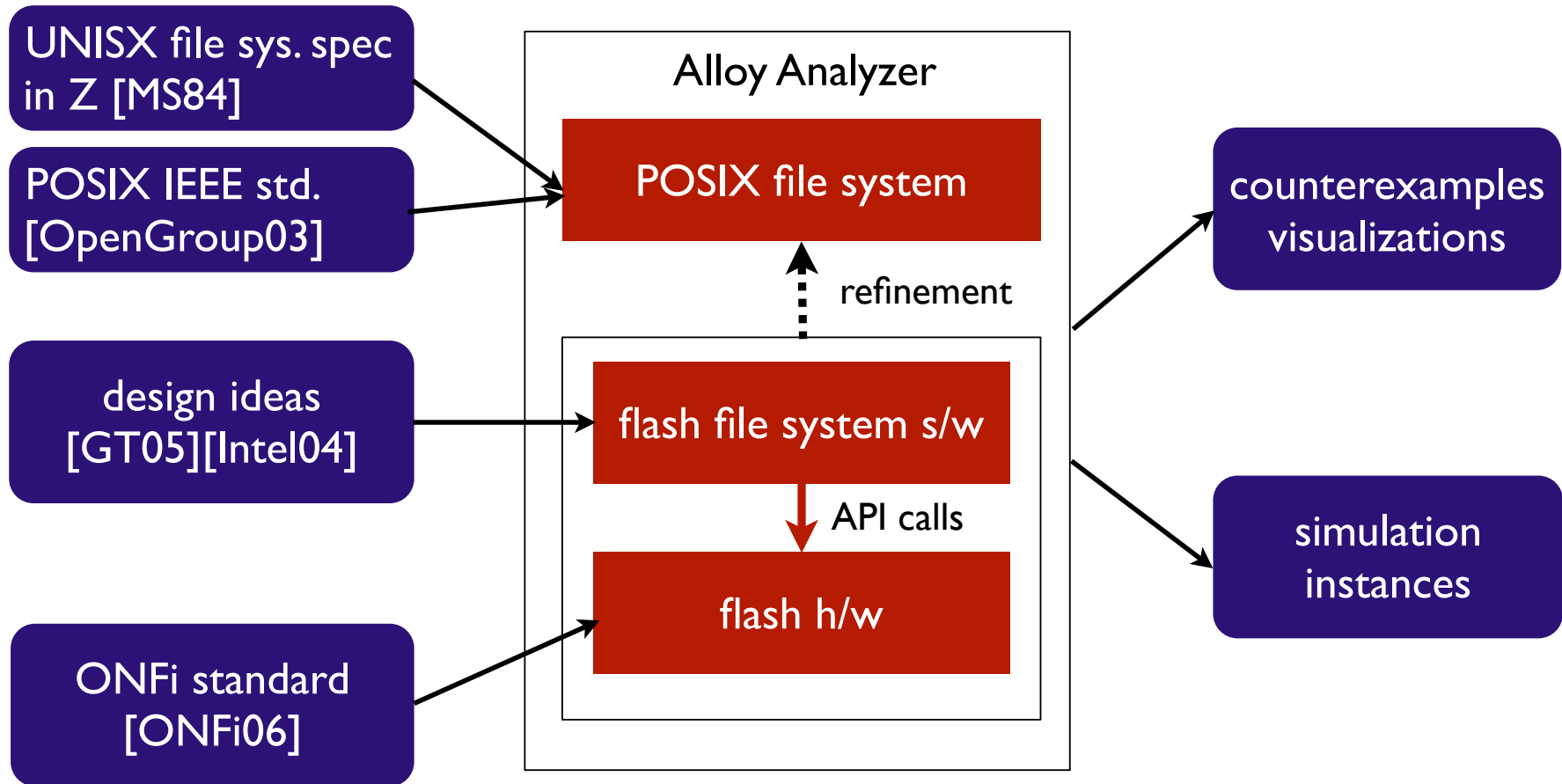
Answer: Formal methods?

- ▶ Allow exhaustive analysis
- ▶ But verifying poorly designed software in an **after-the-fact, ad hoc** manner is impractical

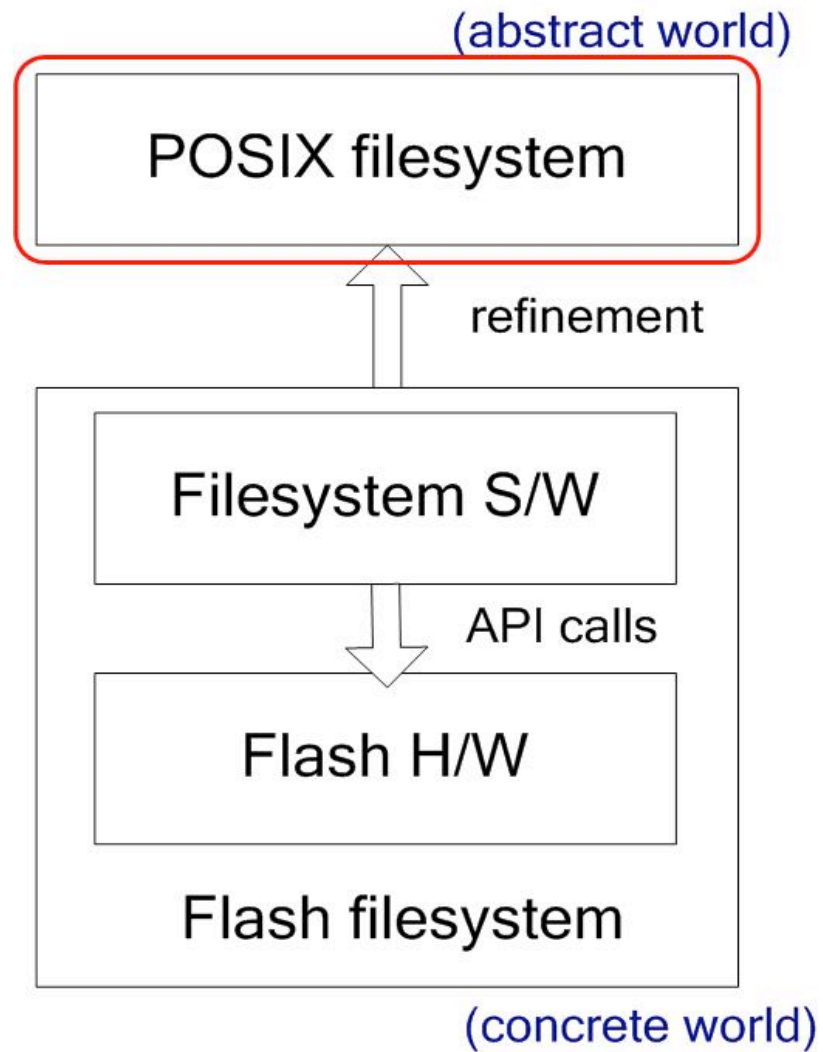
Analyze **early**,
get the **design** right!



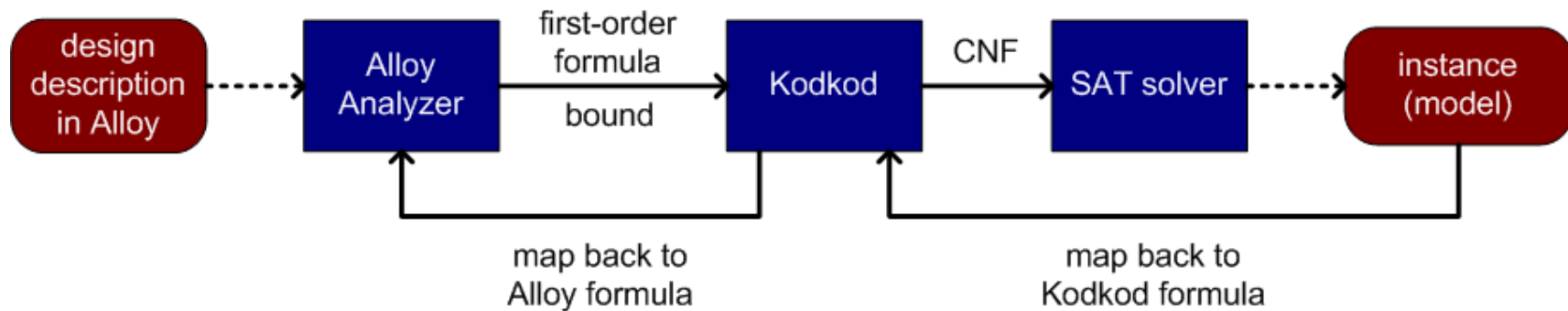
Design approach



POSIX filesystem



Alloy



- ▶ First-order relational logic with transitive closure
- ▶ **Automatic analysis**: simulation & property checking, based on finite model finding (**Kodkod**)

Given system **S**, property **P**, can we find **M**?

$$M \models S \wedge \neg P$$

POSIX file system in Alloy

First Alloy example

```
sig Data {} // data element
sig FID {} // file identifier

sig File {
  contents : seq Data // Int -> Data
}

sig AbsFsys {
  fileMap : FID -> lone File // abstract file system
  // "lone" means one or none
}
```

Abstract write operation

```
pred writeAbs[fsys, fsys' : AbsFsys, fid : FID, buffer : seq Data, offset, size : Int] {  
  let file = fsys.fileMap[fid], file' = fsys'.fileMap[fid],  
      buffer' = buffer.subseq[0, size - 1] {  
    (#buffer' = 0) => file' = file           // case 1  
    (#buffer' != 0) =>                       // case 2 & 3  
      file'.contents = (zeros[offset] ++ file.contents) ++ shift[buffer', offset]  
    promote[fsys, fsys', file', fid]  
  }  
}
```

// promotion

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

Alloy is declarative

- ▶ Like Z, no built-in syntax/semantics for state machines
- ▶ Transition as an explicit constraint between two states

```
pred writeAbs[fsys, fsys' : AbsFsys, fid : FID, buffer : seq Data, offset, size : Int] {  
  let file = fsys.fileMap[fid], file' = fsys'.fileMap[fid],  
      buffer' = buffer.subseq[0, size - 1] {  
    (#buffer' = 0) => file' = file // case 1  
    (#buffer' != 0) => // case 2 & 3  
      file'.contents = (zeros[offset] ++ file.contents) ++ shift[buffer', offset]  
    promote[fsys, fsys', file', fid]  
  }  
}
```

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

Promotion

- ▶ Ensure all other files remain unchanged

```
pred writeAbs[fsys, fsys' : AbsFsys, fid : FID, buffer : seq Data, offset, size : Int] {  
  let file = fsys.fileMap[fid], file' = fsys'.fileMap[fid],  
      buffer' = buffer.subseq[0, size - 1] {  
    (#buffer' = 0) => file' = file           // case 1  
    (#buffer' != 0) =>                       // case 2 & 3  
      file'.contents = (zeros[offset] ++ file.contents) ++ shift[buffer', offset]  
    promote[fsys, fsys', file', fid]  
  }  
}
```

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

(abstract world)

POSIX filesystem

refinement

Filesystem S/W

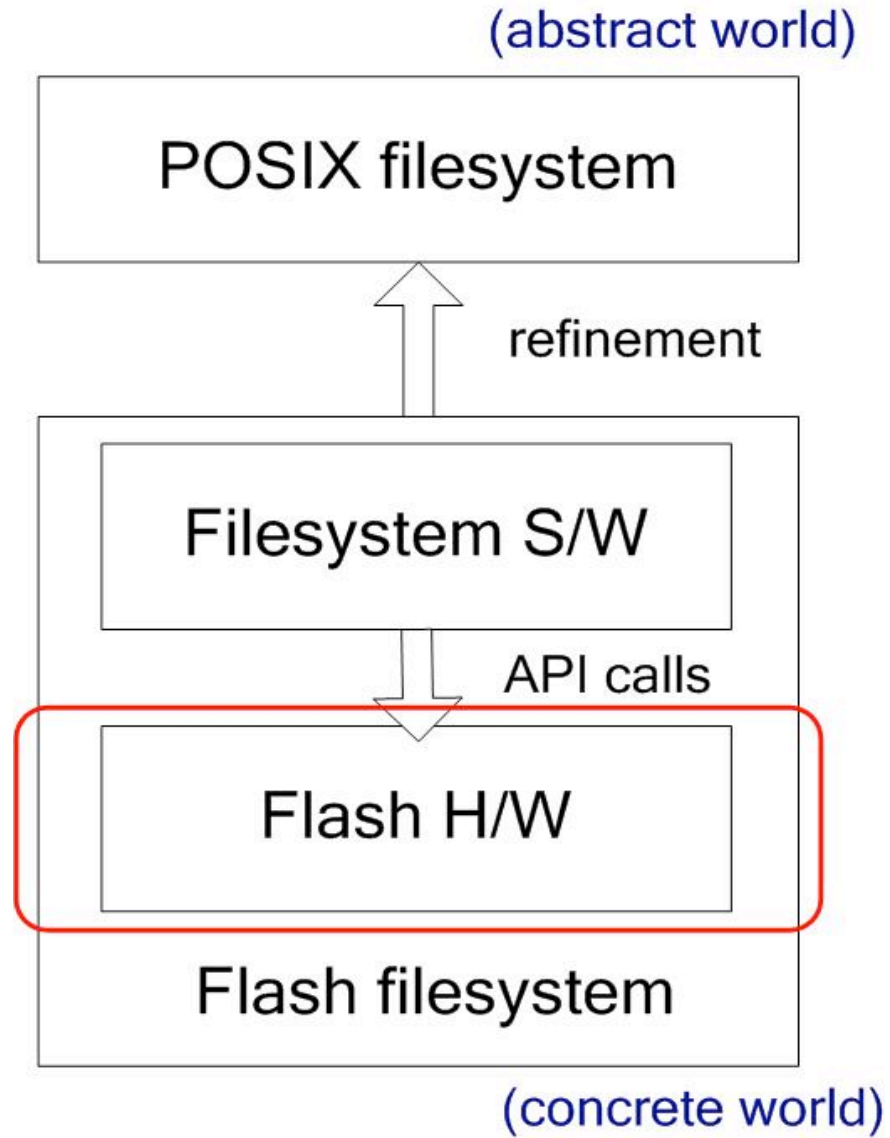
API calls

Flash H/W

Flash filesystem

(concrete world)

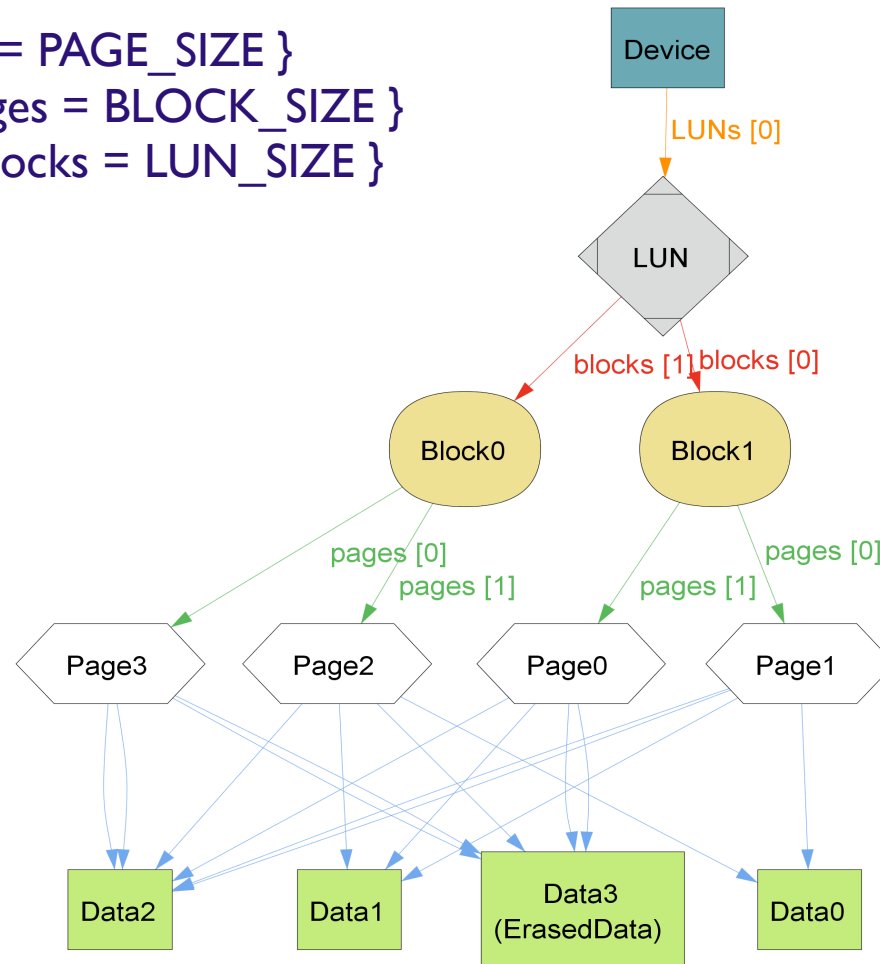
Flash hardware



State of flash hardware

```
sig Page { data : seq Data } { #data = PAGE_SIZE }
sig Block { pages : seq Page } { #pages = BLOCK_SIZE }
sig LUN { blocks : seq Block } { #blocks = LUN_SIZE }
sig Device {
  LUNs : seq LUN
  ...
} { #LUNs = DEVICE_SIZE }
```

```
// simulation with constraints
run {
  some Device
  DEVICE_SIZE = 1
  LUN_SIZE = 2
  BLOCK_SIZE = 2
  PAGE_SIZE = 4
} for 4
```



Operations of flash hardware

// reads data from page

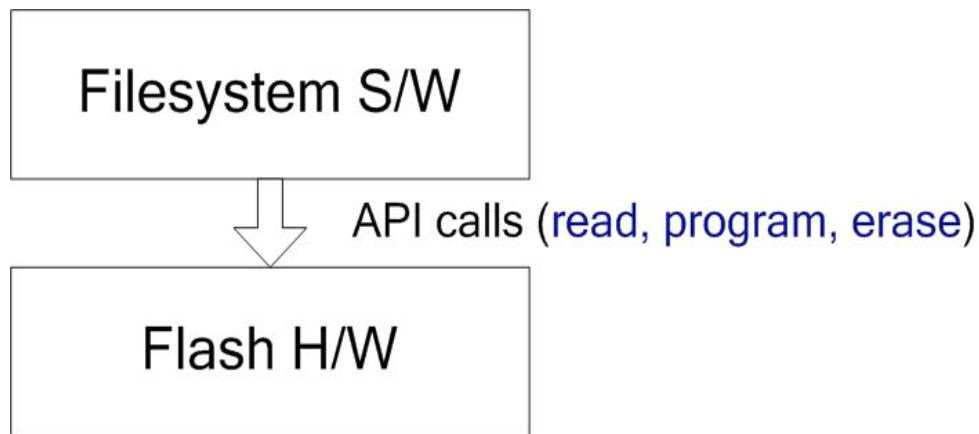
```
fun read[d : Device, colAddr : Int, rowAddr : RowAddr] : seq Data { ... }
```

// program data into page

```
pred program[d, d' : Device, colAddr : Int, rowAddr : RowAddr, data : seq Data] { ... }
```

// erase data in block & increase its erase count

```
pred erase[d, d' : Device, rowAddr : RowAddr] { ... }
```



(abstract world)

POSIX filesystem

refinement

Filesystem S/W

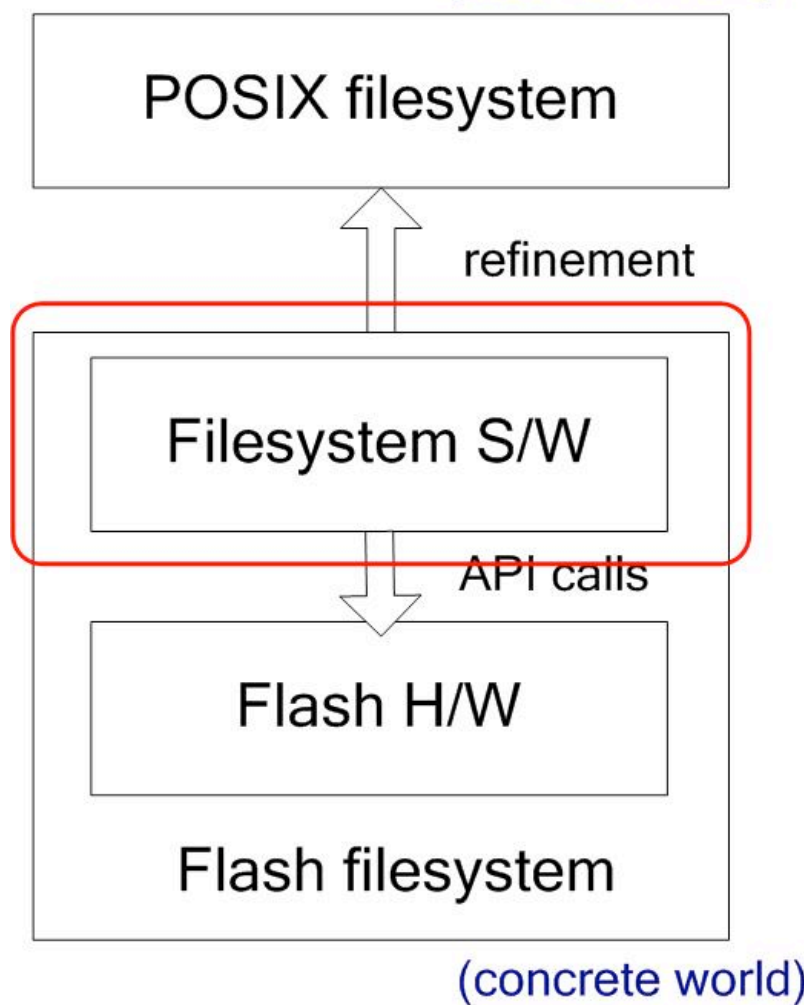
API calls

Flash H/W

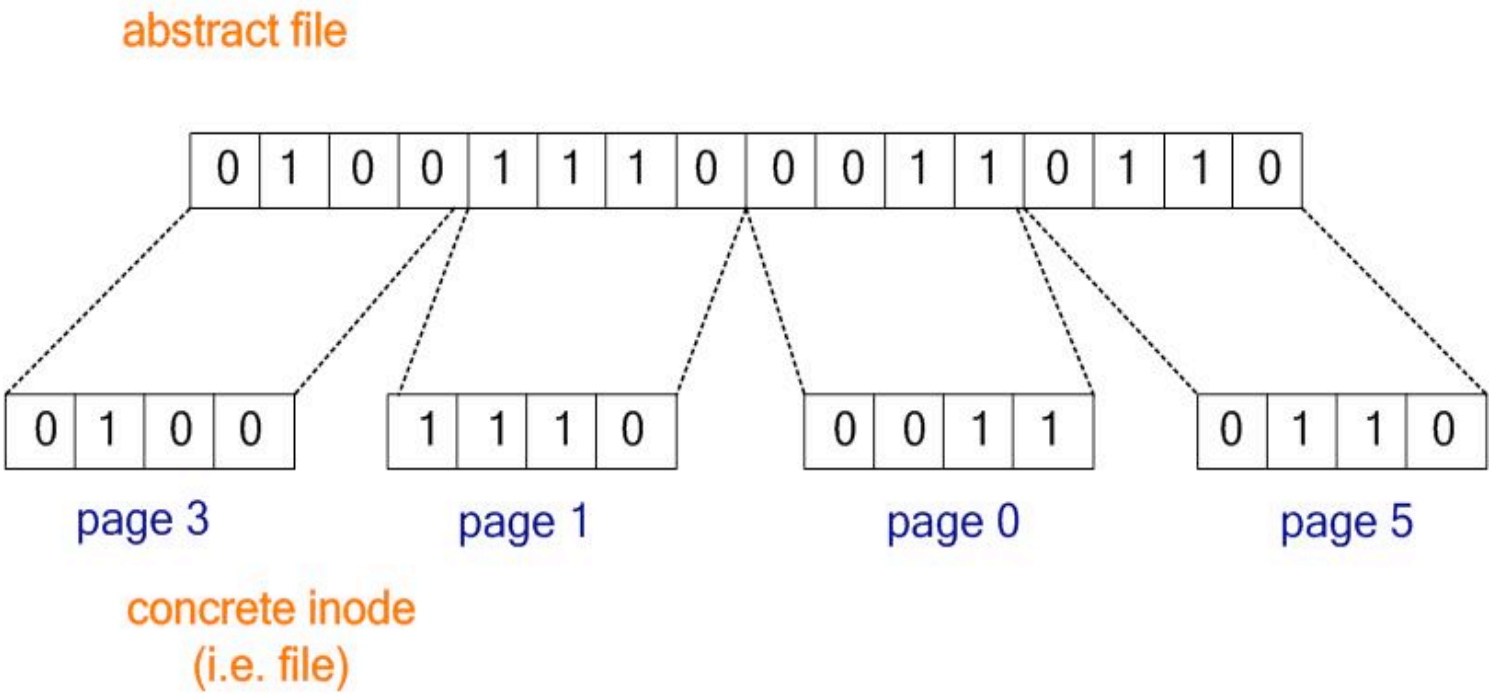
Flash filesystem

(concrete world)

Filesystem software



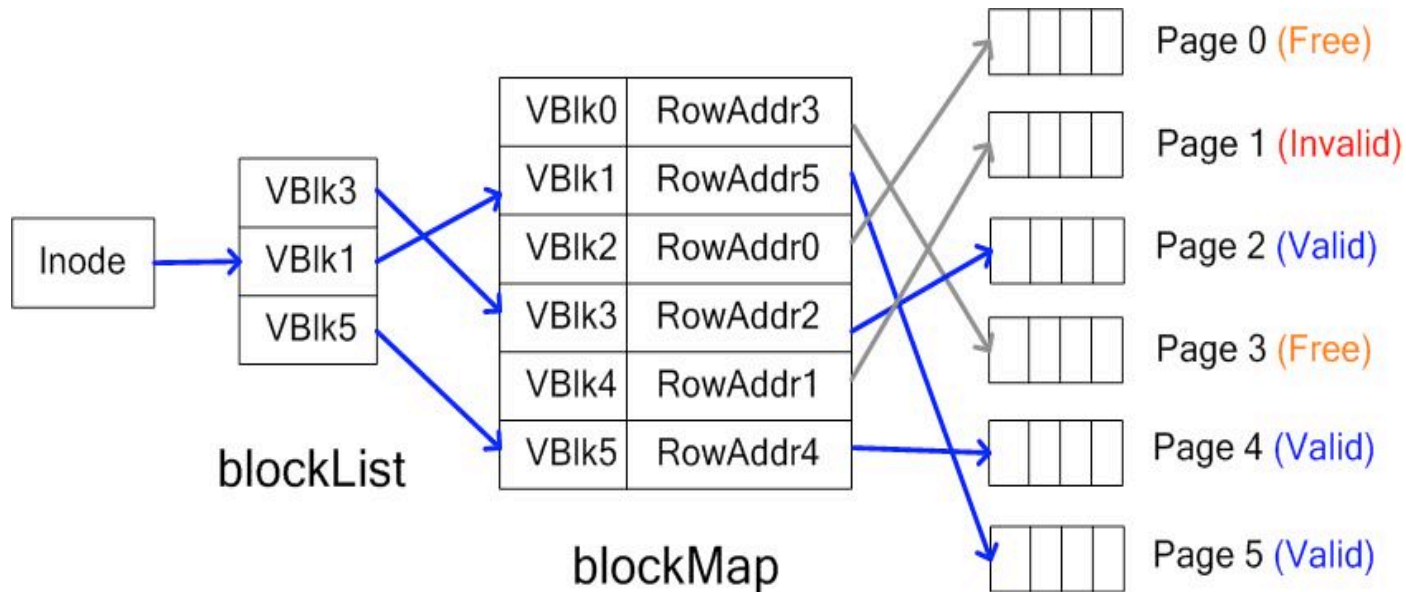
Abstract vs. concrete representation



Concrete file system

```
sig Inode { blockList : seq VBlock }  
sig VBlock { } // virtual block
```

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

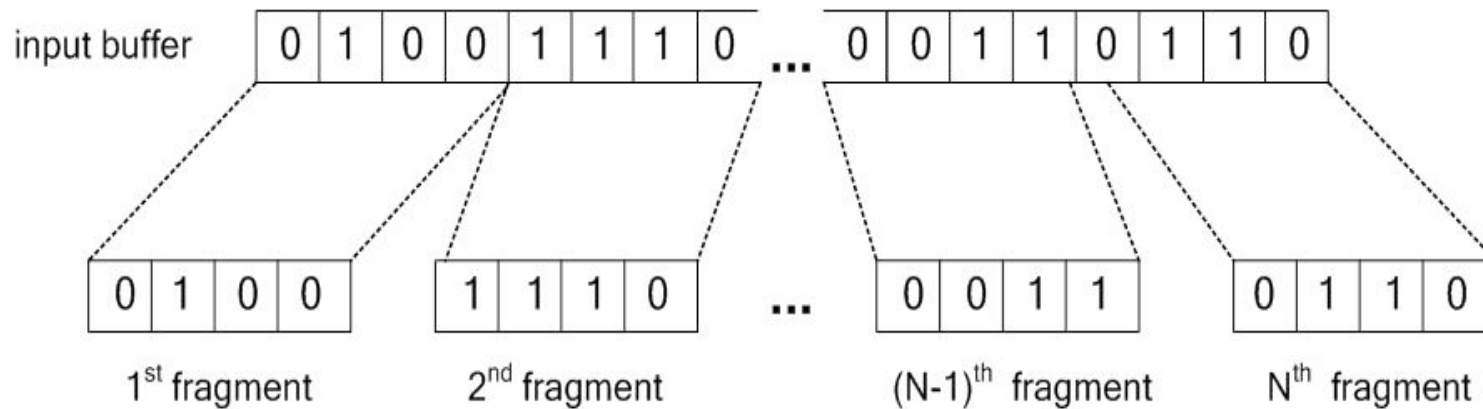


Concrete write operation

Concrete write operation

Basic idea: Break input buffer into fragments & program one by one

e.g. PAGE_SIZE = 4



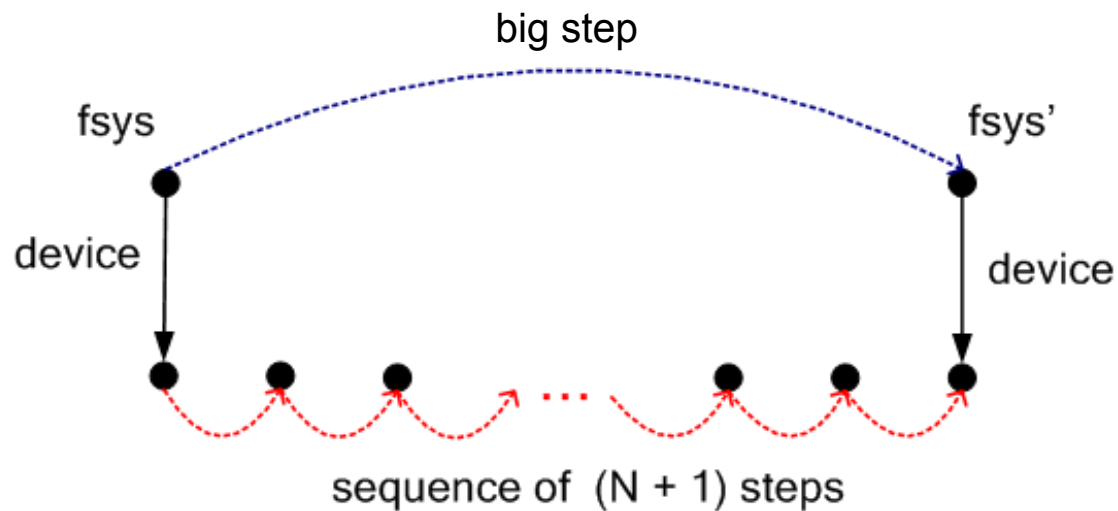
State changes in write operation

File write is a transition between two states f_{sys} and f_{sys}'

$\text{pred writeConc}[f_{sys}, f_{sys}' : \text{ConcFsys}, \text{fid} : \text{FID}, \text{buffer} : \text{seq Data}, \text{offset}, \text{size} : \text{Int}] \{ \dots \}$

But flash program operation is a transition between two device states

$\text{pred program}[d, d' : \text{Device}, \text{colAddr} : \text{Int}, \text{RowAddr} : \text{RowAddr}, \text{data} : \text{seq Data}] \{ \dots \}$



This is different from the standard approach!

Introducing intermediate device states

```
pred writeConc[fsys, fsys' : ConcFsys, fid : FID, buffer : seq Data, offset, size : Int] {  
  ...  
  some stateSeq : seq Device {  
  
    stateSeqConds[fsys.device, fsys'.device, stateSeq, numPagesToProgram]  
  
    all i : stateSeq.butlast.indcs {  
      let from = PAGE_SIZE * i, to = from + PAGE_SIZE - 1,  
          dataFragment = buffer.subseq[from, to],  
          vblock = inode.blockList[startBlkIndex + i],  
          RowAddr = fsys.blockMap[vblock],  
          preState = stateSeq[i], postState = stateSeq[i + 1] |  
          programPage[preState, postState, RowAddr, dataFragment]  
    }  
  }  
  ...  
}
```

stateSeq.first = fsys.device
stateSeq.last = fsys'.device
#stateSeq = numPagesToProgram + 1

Simulating loops in a declarative language

```
pred writeConc[fsys, fsys' : ConcFsys, fid : FID, buffer : seq Data, offset, size : Int] {  
  ...  
  some stateSeq : seq Device {  
  
    stateSeqConds[fsys.device, fsys'.device, stateSeq, numBlocksToProgram]  
  
    all i : stateSeq.butlast.inds {  
      let from = PAGE_SIZE * i, to = from + PAGE_SIZE - 1,  
          dataFragment = buffer.subseq[from, to],  
          vblock = inode.blockList[startBlkIndex + i],  
          RowAddr = fsys.blockMap[vblock],  
          preState = stateSeq[i], postState = stateSeq[i + 1] |  
              programPage[preState, postState, rowAddr, dataFragment]  
    }  
  }  
  ...  
}
```

quantified variable as loop index

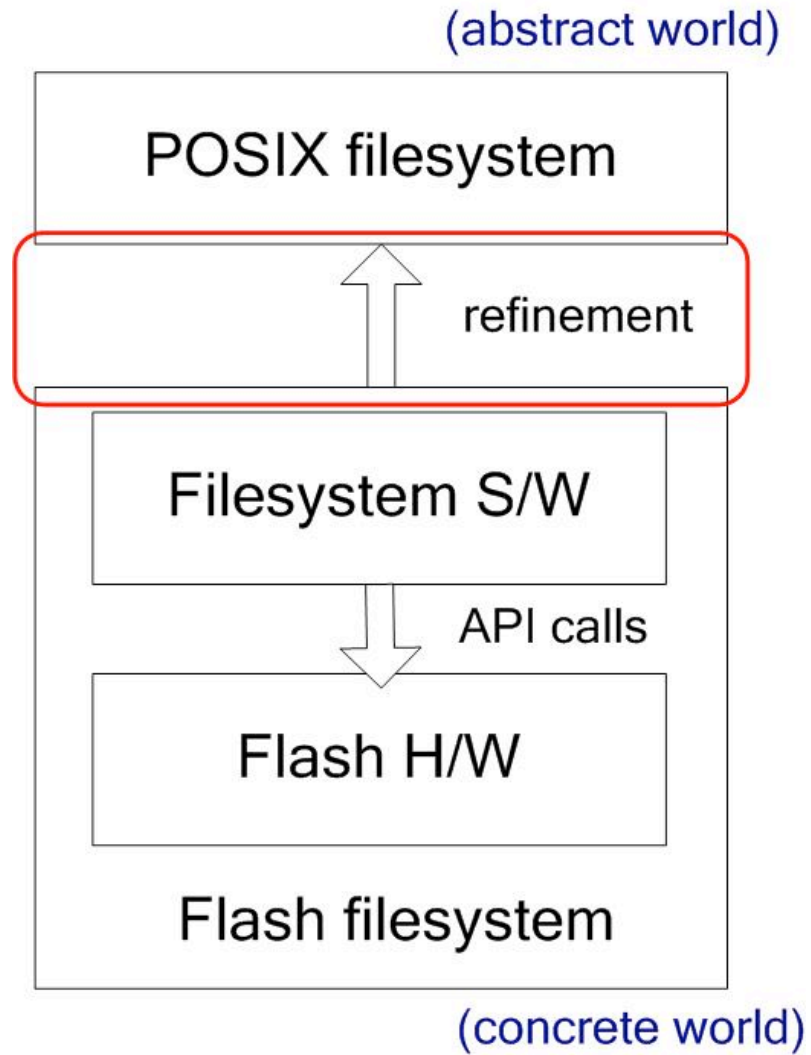
Write operation (snippet)

```
pred writeConc[fsys, fsys' : ConcFsys, fid : FID, buffer : seq Data, offset, size : Int] {  
  ...  
  some stateSeq : seq Device {  
    stateSeqConds[fsys.device, fsys'.device, stateSeq, numBlocksToProgram]  
    all i : stateSeq.butlast.indxs {  
      let from = PAGE_SIZE * i, to = from + PAGE_SIZE - 1  
      dataFragment = buffer.subseq[from, to],  
      vblock = inode.blockList[startBlkIndex + i],  
      RowAddr = fsys.blockMap[vblock],  
      preState = stateSeq[i], postState = stateSeq[i + 1] |  
      programPage[preState, postState, rowAddr, dataFragment]  
    }  
  }  
  ...  
}
```

What else goes on here?

- **Wear-leveling**
- **Garbage collection**
- **Fault-tolerance**

Refinement

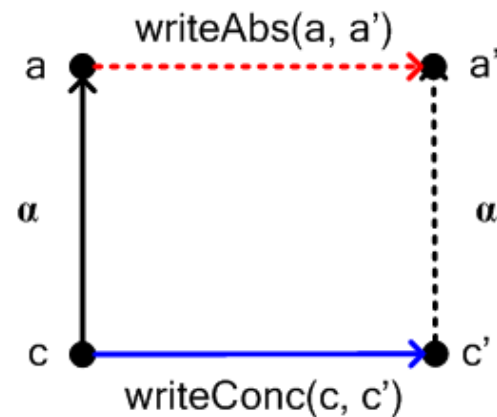


Data Refinement

Does the concrete system conform to the abstract system?

e.g. write operation

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



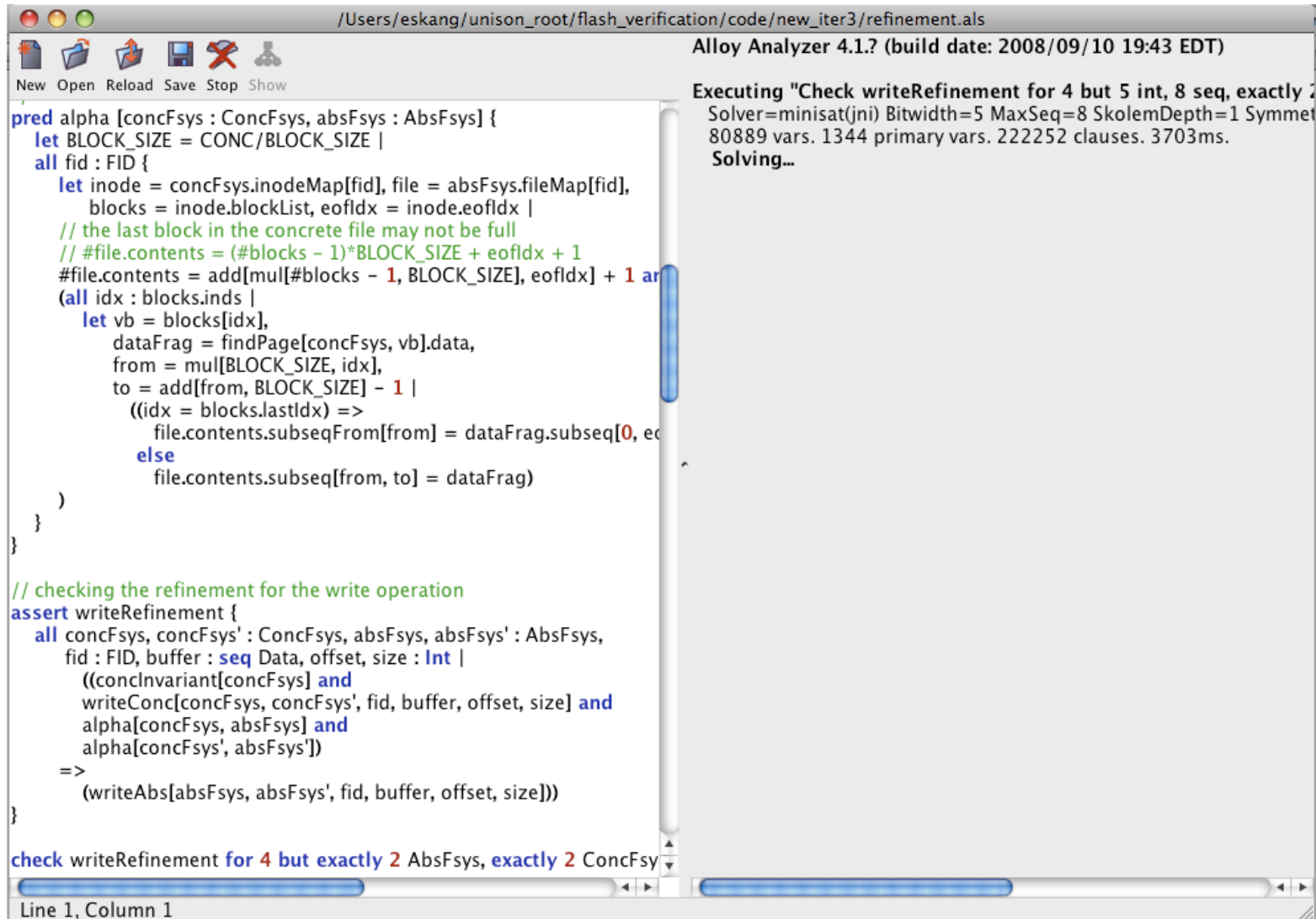
Analysis results

WriteRefinement:

- ▶ A scope of 5 for each domain
(except: 6 pages, each with 4 data elements)
- ▶ Incremental modeling & analysis
- ▶ Found **~20 bugs** over multiple iterations
- ▶ Final version returned no counterexample, **~8 hours** to check

Example

Checking write operation



The screenshot displays the Alloy Analyzer 4.1.7 interface. The left pane shows the Alloy code for a refinement check, and the right pane shows the execution output.

```
pred alpha [concFsys : ConcFsys, absFsys : AbsFsys] {
  let BLOCK_SIZE = CONC/BLOCK_SIZE |
  all fid : FID {
    let inode = concFsys.inodeMap[fid], file = absFsys.fileMap[fid],
        blocks = inode.blockList, eofIdx = inode.eofIdx |
    // the last block in the concrete file may not be full
    // #file.contents = (#blocks - 1)*BLOCK_SIZE + eofIdx + 1
    #file.contents = add[mul[#blocks - 1, BLOCK_SIZE], eofIdx] + 1 and
    (all idx : blocks.inds |
      let vb = blocks[idx],
          dataFrag = findPage[concFsys, vb].data,
          from = mul[BLOCK_SIZE, idx],
          to = add[from, BLOCK_SIZE] - 1 |
      ((idx = blocks.lastIdx) =>
        file.contents.subseqFrom[from] = dataFrag.subseq[0, eofIdx]
      else
        file.contents.subseq[from, to] = dataFrag)
    )
  }
}

// checking the refinement for the write operation
assert writeRefinement {
  all concFsys, concFsys' : ConcFsys, absFsys, absFsys' : AbsFsys,
  fid : FID, buffer : seq Data, offset, size : Int |
  ((conclnvariant[concFsys] and
   writeConc[concFsys, concFsys', fid, buffer, offset, size] and
   alpha[concFsys, absFsys] and
   alpha[concFsys', absFsys'])
 =>
  (writeAbs[absFsys, absFsys', fid, buffer, offset, size]))
}

check writeRefinement for 4 but exactly 2 AbsFsys, exactly 2 ConcFsys
```

Alloy Analyzer 4.1.7 (build date: 2008/09/10 19:43 EDT)

Executing "Check writeRefinement for 4 but 5 int, 8 seq, exactly 2 AbsFsys, exactly 2 ConcFsys"
Solver=minisat(jni) Bitwidth=5 MaxSeq=8 SkolemDepth=1 Symmet
80889 vars. 1344 primary vars. 222252 clauses. 3703ms.
Solving...

Line 1, Column 1

Counterexample found!

The screenshot shows the Alloy Analyzer 4.1.7 interface. The top window title is `/Users/eskang/unison_root/flash_verification/code/new_iter3/refinement.als`. The top right corner displays `Alloy Analyzer 4.1.7 (build date: 2008/09/10 19:43 EDT)`. The main window is split into three panes:

- Code Editor (Left):** Contains a Z3-style predicate definition for `alpha`. The code includes a `pred alpha` block with parameters `[concFsys : ConcFsys, absFsys : AbsFsys]`. It defines `BLOCK_SIZE`, iterates over `fid` and `blocks`, and performs a `writeConc` operation. A comment `// checking the refinement for the write operation` is present above an `assert writeRefinement` block. The `assert` block contains an `all` quantifier over `concFsys`, `concFsys'`, `absFsys`, and `absFsys'`, with conditions for `writeConc` and `alpha`. The `check` command at the bottom is `check writeRefinement for 4 but exactly 2 AbsFsys, exactly 2 ConcFsys`.
- Command Line (Right):** Shows the execution of the `writeRefinement` check. The command is `Executing "Check writeRefinement for 4 but 5 int, 8 seq, exactly 2 AbsFsys, exactly 2 ConcFsys"`. The solver used is `minisat(jni)` with parameters `Bitwidth=5 MaxSeq=8 SkolemDepth=1 Symmetry=20`. The output is `80889 vars. 1344 primary vars. 22252 clauses. 3703ms. Counterexample found. Assertion is invalid. 94498ms.`
- Status Window (Bottom):** Displays the same execution details as the command line. The word `Counterexample` is circled in red.

The status window text is: `Executing "Check writeRefinement for 4 but 5 int, 8 seq, exactly 2 AbsFsys, exactly 2 ConcFsys", Solver=minisat(jni) Bitwidth=5 MaxSeq=8 SkolemDepth=1 Symmetry=20 80889 vars. 1344 primary vars. 22252 clauses. 3703ms. Counterexample found. Assertion is invalid. 94498ms.`

Counterexample visualization

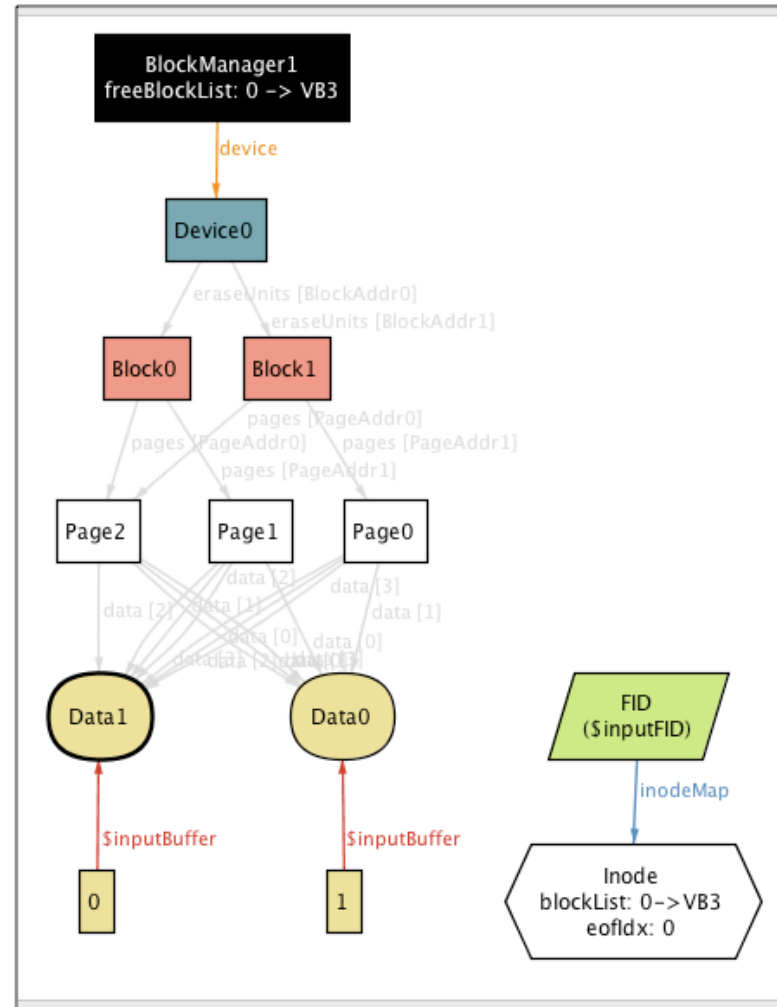
Operation:

Write [Data1, Data0] to the inode identified by FID

Expected:

[Data0] \Rightarrow [Data1, Data0]

Pre-state of concrete system



How did the concrete system violate the spec?

Operation:

Write [Data1, Data0] to the inode identified by FID

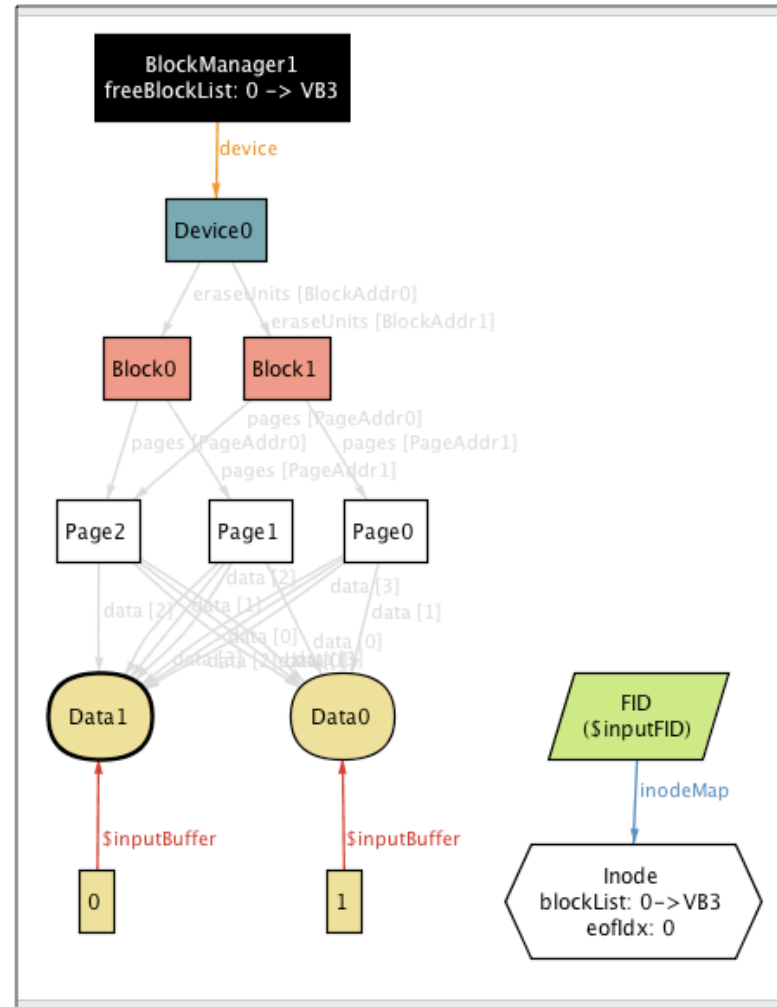
Expected:

[Data0] ⇒ [Data1, Data0]

Observed:

[Data0] ⇒ [Data0]

In fact, the inode did not change at all!



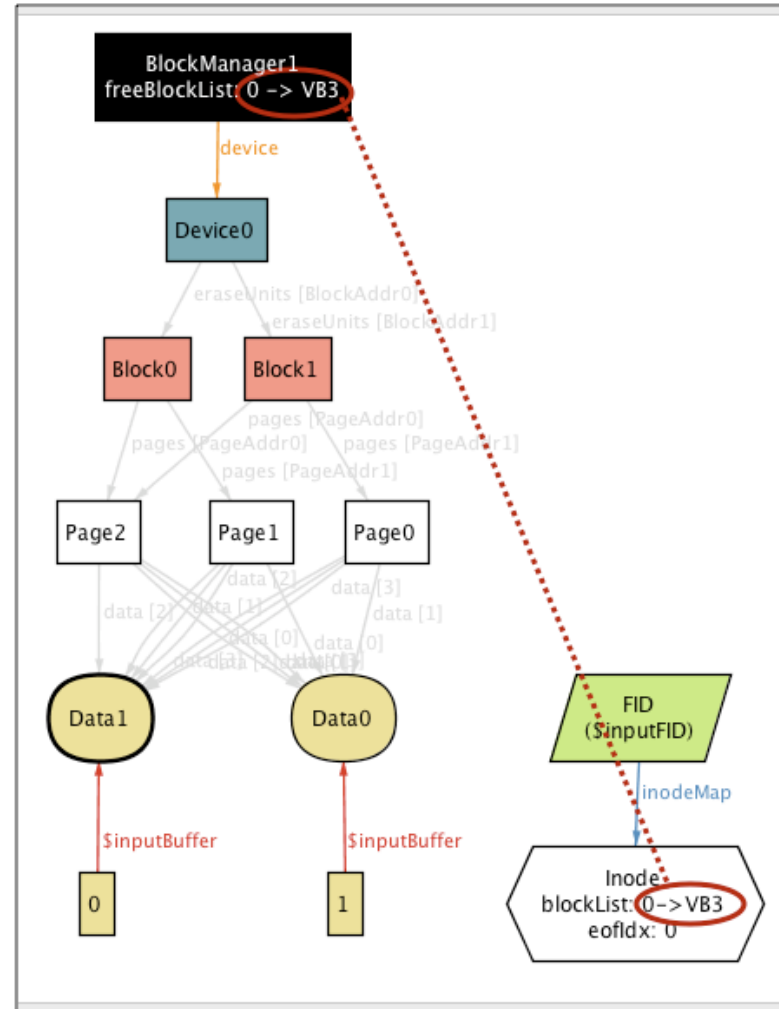
Invalid pre-state!

Problem:

A block in the list of free blocks is already in use by the inode

Fix:

Strengthen the invariant



Strengthening the invariant

Problem:

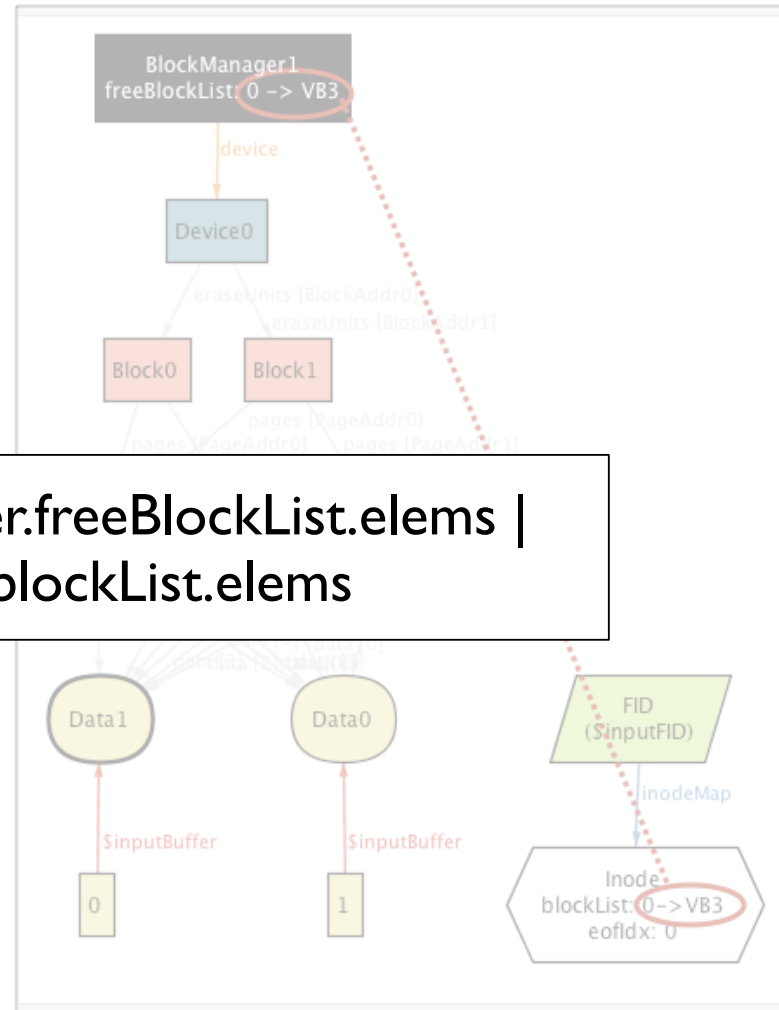
A block in the list of free blocks is already in use by the inode

Fix:

Strengthen the invariant

all freeBlock : blockManager.freeBlockList.elems | freeBlock **not in** Inode.blockList.elems

Invariant preservation can be checked separately



Discussion

Discussion

On analysis:

- ▶ Properties analyzed only for small instances of file system
- ▶ Many subtle errors occur in “boundary” cases, involving a small number of components
- ▶ **Complementary** to other approaches:
Find bugs using bounded analysis, and then prove (or vice-versa)

On language:

- ▶ Being declarative is nice, but not always convenient
- ▶ Better syntax for sequential composition and loops?

Future work:

- ▶ Extended functionality (directories, etc.)
- ▶ Implementation
- ▶ Test-case generation (using SAT enumeration)