Language
module addressBook

sig Name, Addr {}

sig Book {
  addr: Name -> lone Addr
}

pred show[] { }

run show for 3 but 1 Book

our first model
(pause for demo)
module tour/addressBook1a ---- Page 6

sig Name, Addr {}

sig Book {
    addr: Name -> lone Addr
}

pred show {}

// This command generates an instance similar to Fig 2.1
run show for 3 but 1 Book

Executing "Run show for 3 but 1 Book"
Solver=minisat(jni) Bitwidth=4 MaxSeq=3 SkolemDepth=1 Symmetry=20 153 vars. 16 primary vars. 219 clauses. 162ms.
Instance found. Predicate is consistent. 11ms.
Executing "Run show for 3 but 1 Book"
Solver=minisat(jni) Bitwidth=4 MaxSeq=3 SkolemDepth=1 Symmetry=20
153 vars. 16 primary vars. 219 clauses. 162ms.
Instance found. Predicate is consistent. 11ms.

(addrBook1a) Run show for 3 but 1 Book

addr: 1

Book

addr [Name]

Addr   Name
module addressBook

open util/ordering[Book] as ord

module util/ordering[exactly elem]

modules
open util/ordering[Book] as ord
open util/ordering[Name] as nameOrd

ord/ first
ameOrd/first

importing “as”
module util/ordering[exactly elem]

private one sig Ord {
    First: set elem,
    Next: elem -> elem
}

pred/totalOrder[elem,First,Next]
sig Name, Addr {}
sig Book {
    addr: Name -> lone Addr
}

signatures
signature

A

{ }

A

module

addressBook

open util/ordering[Book] as order

open util/ordering[Name] as nameOrder

module util/ordering[ExactlyElem]

private

one

sig

Ord

{

First: set elem,

Next: elem − elem

}

pred partialOrder[elem,First,Next]

run p for 4

run p for 3 but 2

assert a { ... }

check a for 4 but 1

sig

NoDe

{

left, right: LoneNoDe

}

fun

leftSubTree[n: NoDe]: set NoDE

{ 2 }
sig A, B {}
sig A, B {}
sig A1, A2 extends A {}
abstract sig A {}
sig A1, A2 extends A {}
private sig A {}
sig Name, Addr {}
sig Book {
    addr: Name -> lone Addr
}

fields
sig Name, Addr {}

sig Book {
    addr: Name → lone Addr
}

addr : Book→Name→Addr

fields (are relations)
Exercise 1

Write a new address book model whose metamodel has this structure.
pred show[] { 
  #b.addr > 1 
  #Name.(b.addr) + 1 
}

predicates
pred del[b,b': Book, n: Name] {
  b'.addr = b.addr - n->Addr
}

relational join
relational join

\[
\begin{align*}
&\{ (a,b,c) \\
&\{ (e, f, g) \\
&\{ (h, i, j) \}
\} \quad \cdot \quad \{ \\
&\{ (c, e, f, z) \\
&\{ (c, b, a, u) \\
&\{ (g, z, a, b) \}
\} \\
\}
\end{align*}
\]

= \{ \\
&\{ (a, b, e, f, z) \\
&\{ (e, f, z, a, b) \\
&\{ (a, b, b, a, u) \}
\} \\
\}

relational join
fun lookup[b: Book, n: Name] {
    n.(b.addr)
}

addr : Book→Name→Addr
b.addr : Name→Addr
n.(b.addr) : Addr

relational join
pred show[] {} 
run show for 3 but 1 Book

assert a {} ...
check a for 4 but 1 A ...

commands & assertions
pred show[] {
    #b.addr > 1
    #Name.(b.addr) + 1
}

cardinality
\textbf{pred} add[b,b': Book, n: Name, a: Addr] \{ 
  \textcolor{red}{b'.addr = b.addr + n\rightarrow a}
\}

\textbf{pred} del[b,b': Book, n: Name] \{ 
  b'.addr = b.addr - n\rightarrow Addr
\}

dynamics
assert delUndoesAdd {
  all  b,b’,b ’’:  Book, n:  Name,  a:  Addr |
  no  n.(b.addr) &&
      add[b,b’,n,a] && del[b’,b ’’, n] => b.addr = b’’.addr
}
check delUndoesAdd for 10 but 3 Book
assert delUndoesAdd {
    all b,b’,b ”: Book, n: Name, a: Addr |
    no n.(b.addr) &&
    add[b,b’,n,a] && del[b’,b ”, n] => b.addr = b”.addr
}
cHECK delUndoesAdd for 10 but 3 Book
\textbf{transitive closure}
transitive closure

N1 \^left = \{N2, N4, N6\}

left = \{(N1, N2),(N2, N4),(N4, N6)\}

\^left = \{(N1, N2),(N2, N4),(N4, N6),(N1, N4),(N1, N6),(N2, N6)\}
**reflexive transitive closure**

\[ *\text{left} = ^{\wedge}\text{left} + \text{iden} \]

\[ N1.*\text{left} = \{N1, N2, N4, N6\} \]

\[ *\text{left} = \]

\[
\begin{align*}
(N1, N2) \\
(N2, N4) \\
(N4, N6) \\
(N1, N4) \\
(N2, N6) \\
(N1, N6)
\end{align*}
\]

\[ + \]

\[
\begin{align*}
(N1, N1) \\
(N2, N2) \\
(N3, N3) \\
(N4, N4) \\
\ldots \\
(B1, B1) \\
\ldots
\end{align*}
\]

reflexive transitive closure
relations, sets, and “scalars”

\{(a,b,c), (d,e,f), (g,h,i)\}

\{a,b,c\} = \{(a), (b), (c)\}

\(g = \{(g)\}\)
$N1.\ ^{\text{left}}$  
\[
\{ (N1) \} \cdot \{ (N2), (N4), (N6) \} = \{ (N2), (N4), (N6) \}
\]
relations, sets, and “scalars”
Exercise 2

Write **lookup**, **add**, and **del** predicates for the new address book.
Analysis
Types of Analysis

Simulation

Checking assertions
Ex: Telephone Exchange System
Before we begin...

Download the model here:

http://people.csail.mit.edu/eskang/alloy_tutorial
Basic Model

module telephone_switch

sig Number {}

sig Phone {}

sig Switch {
  mapping : Phone lone -> Number, 
  connected : Phone lone -> lone Phone
}

Model Finding

Given a formula $F$, find a model $M$ such that $M \models F$
Analysis I: Simulation
Simulation

Given a system specification $S$, find a model $M$ such that $M \models S$

$M$ is a simulation instance of $S$
Simulating Telephone Switch

sig Number {}
sig Phone {}
sig Switch {
    mapping : Phone lone -> Number,
    connected : Phone lone -> lone Phone
}
run {} for 3
Simulating Telephone Switch

```plaintext
sig Number {}
sig Phone {}
sig Switch {
  mapping : Phone lone -> Number,

  connected : Phone lone -> lone Phone
}
run {} for 3
```
Generating Interesting States

Specify additional constrains inside run command

sig Number {}
sig Phone {}
sig Switch {
  mapping : Phone lone -> Number,
  connected : Phone lone -> lone Phone
}
run {
  some mapping
  some connected
} for 3
sig Number {};
sig Phone {};
sig Switch {
  mapping : Phone lone -> Number,
  connected : Phone lone -> lone Phone
}
run {
  some mapping
  some connected
} for 3
Customizing Visualization

sig Number {}

sig Phone {}

sig Switch {
    mapping: Phone lone -> Number,
    connected: Phone lone -> lone Phone
}

run {
    some mapping
    some connected
} for 3
Customizing Visualization

sig Number {}

sig Phone {}

sig Switch {
  mapping : Phone lone -> Number,
  connected : Phone lone -> lone Phone
}

run {
  some mapping
  some connected
} for 3

Is this a good state?
Simulation with State Invariant

Use predicates to define state invariants

pred noDanglingConnection[s : Switch] {
  all p : Phone |
  p in (s.connected[Phone] + s.connected.Phone) implies
  p in s.mapping.Number
}
Simulation with State Invariant

Use predicates to define state invariants

pred noSelfConnection[s : Switch] {
    no (iden & s.connected)
}
Simulation with State Invariant

pred invariant[s : Switch] {  
    noDanglingConnection[s]  
    noSelfConnection[s]  
}

run {  
    some s : Switch |  
    invariant[s] and  
    some mapping and  
    some connected  
} for 3 but 1 Switch
Simulation with State Invariant

pred invariant{s : Switch} {
  noDanglingConnection[s]
  noSelfConnection[s]
}

run {
  some s : Switch |
  invariant[s] and
  some mapping and
  some connected
} for 3 but 1 Switch
Simulating an Operation

Establish a connection from caller to phone at dialed

pred requestCall[s, s' : Switch, caller : Phone, dialed : Number] {
  s'.connected = s.connected + (caller --> s.mapping.dialed)
}

run RequestCall {
  some s, s' : Switch, p : Phone, n : Number |
  invariant[s] and
  requestCall[s, s', p, n]
} for 3 but 2 Switch
Establish a connection from caller to phone at dialed

pred requestCall[s, s': Switch, caller : Phone, dialed : Number] {
  s'.connected = s.connected + (caller -> s.mapping.dialed)
}
run RequestCall {
  some s, s': Switch, p : Phone, n : Number |
  invariant[s] and
  requestCall[s, s', p, n] and
  s != s'
} for 3 but 2 Switch
Simulating an Operation

Establish a connection from caller to phone at dialed

pred requestCall[s, s' : Switch, caller : Phone, dialed : Number] {
  s'.connected = s.connected + (caller -> s.mapping.dialed)
}

run RequestCall {
  some s, s' : Switch, p : Phone, n : Number |
  invariant[s] and
  requestCall[s, s', p, n] and
  s != s'
} for 3 but 2 Switch

What is missing?
Frame Conditions

Fix parts of the state that do not change!

pred requestCall[s, s' : Switch, caller : Phone, dialed : Number] { 
  s'.connected = s.connected + (caller -> s.mapping.dialed) 
  s'.mapping = s.mapping 
}

run RequestCall { 
  some s, s' : Switch, p : Phone, n : Number | 
  invariant[s] and 
  requestCall[s, s', p, n] 
} for 3 but 2 Switch
Recognizing Invalid Operations

What is wrong with this scenario?

```
pred requestCall[s, s' : Switch, caller : Phone, dialed : Number] {
  s'.connected = s.connected + (caller -> s.mapping.dialed)
  s'.mapping = s.mapping
}
run RequestCall {
  some s, s' : Switch, p : Phone, n : Number |
  invariant[s] and
  requestCall[s, s', p, n]
} for 3 but 2 Switch
```

pre-state = post-state
Disallowing Invalid Operations

A phone must exist at the dialed number!

\[
\text{sig Switch} \{
\text{mapping} : \text{Phone lone} \rightarrow \text{Number},
\text{connected} : \text{Phone lone} \rightarrow \text{lone Phone}
\}
\]

\[
\text{pred requestCall}[s, s' : \text{Switch}, \text{caller} : \text{Phone}, \text{dialed} : \text{Number}] \{ \\
some s.\text{mapping.dialed} \\
s'.\text{connected} = s.\text{connected} + (\text{caller} \rightarrow s.\text{mapping.dialed}) \\
s'.\text{mapping} = s.\text{mapping}
\}
\]

Like a precondition; if it does not hold, then
the operation does not occur (i.e. pred. is false)
Invalid Operation: Another Ex.

What is wrong with this scenario?

```java
pred requestCall[s, s': Switch, caller : Phone, dialed : Number] {
    some mapping.dialed
    s'.connected = s.connected + (caller -> s.mapping.dialed)
    s'.mapping = s.mapping
}
```
Refining the Operation

Should not allow a self-referencing call!

```plaintext
pred requestCall[s, s' : Switch, caller : Phone, dialed : Number] { 
  some s.mapping.dialed
  caller != s.mapping.dialed
  s'.connected = s.connected + (caller -> s.mapping.dialed)
  s'.mapping = s.mapping
}
```
Simulation-Driven Refinement

You will most likely to be wrong first time!

Repeat the Run-Visualize-Refine cycle

Even tiny instances can reveal flaws and insights
Exercise 1: Hang Up Operation

Terminate the connection involving caller

pred hangUp[s, s' : Switch, caller : Phone] {
  // fill in the details here
  ...

}
pred requestCall[s, s' : Switch, caller : Phone, dialed : Number] { 
    // a phone at the dialed number exists 
    some s.mapping.dialed 
    // caller cannot call itself 
    caller != s.mapping.dialed 
    // callee is not already connected 
    not s.mapping.dialed in 
        (s.connected[Phone] + s.connected.Phone) 
    s'.connected = s.connected + (caller -> s.mapping.dialed) 
    s’.mapping = s.mapping 
}
Analysis II: Checking Assertions
Checking Assertions

Given a system specification $S$ and a property $P$, find a model $M$ such that $M \models S \land \neg P$

$M$ is a counterexample to $P$
Checking Invariant Preservation

Make sure an operation preserves the invariant

Use **assert** to state a property, and **check** to analyze it

```assert```
```
RequestCallPreservesInvariant {
    all s, s' : Switch, p : Phone, n : Number |
        invariant[s] and requestCall[s, s', p, n] implies
        invariant[s']
}
```

```
check```
```
RequestCallPreservesInvariant for 3 but 2 Switch
Scope-Complete Analysis

Exhaustive up to the bounds on signatures
Small-Scope Hypothesis

Absence of a counterexample ➠ the property holds

But most flaws have small counterexamples

Increase scope for higher confidence
Specifying Bounds

```求婚 RegionCallPreservesInvariant {
    all s, s' : Switch, p : Phone, n : Number |
    invariant[s] and requestCall[s, s', p, n] implies
    invariant[s']
} ```

```求婚 RequestCallPreservesInvariant for 3 but 2 Switch

Each counterexample can have...
...up to 2 Switch atoms
...and up to 3 atoms of every other signature
assert RequestCallPreservesInvariant { all s, s': Switch, p : Phone, n : Number | invariant[s] and requestCall[s, s', p, n] implies invariant[s'] }
Page 1:

Counterexample Found

```
assert RequestCallPreservesInvariant { all s, s', p : Switch, n : Number | invariant[s] and requestCall[s, s', p, n] implies invariant[s'] }
```

```
check RequestCallPreservesInvariant for 3 but 2 Switch
```

The caller doesn’t have a phone number! It violates the invariant `noDanglingConnection`
Fixing the Bug

pred requestCall[s, s' : Switch, caller : Phone, dialed : Number] {
    // caller must have a number
    some s.mapping[caller]

    // a phone at the dialed number exists
    some s.mapping.dialed

    // caller cannot call itself
    caller != s.mapping.dialed

    // callee is not already connected
    not s.mapping.dialed in
    (s.connected[Phone] + s.connected.Phone)

    s'.connected = s.connected + (caller -> s.mapping.dialed)
    s’.mapping = s.mapping
}

Re-Running the Analysis

**check** RequestCallPreservesInvariant for 3 but 2 Switch

No more counterexample!
(same for larger bounds)
Are We Done?

“...the purpose of verification is not to produce peace of mind, but to find bugs in programs...the message at the successful exit of program verifiers should be changed from “Verified” to “Sorry, can’t find any more errors.””

Back to Simulation

Let us try a few more scenarios

run RequestCall {
    some s, s' : Switch, p : Phone, n : Number |
    invariant[s] and
    requestCall[s, s', p, n]
} for 3 but 2 Switch
Back to Simulation

Returns the next instance
Back to Simulation

What's wrong here?
A phone can’t be in multiple connections simultaneously!

pred noSimultaneousConnections[s : Switch] { all p : Phone | lone (s.connected[p] + s.connected.p) }
pred invariant[s : Switch] { noSimultaneousConnections[s] noDanglingConnection[s] noSelfConnection[s] }
Re-Running the Analysis

assert RequestCallPreservesInvariant {
  all s, s', p : Switch, n : Number |
  invariant[s] and requestCall[s, s', p, n] implies invariant[s']
}

check RequestCallPreservesInvariant for 3 but 2 Switch

The same scenario is now a counterexample!
Fixing the Bug

def requestCall(s, s': Switch, caller: Phone, dialed: Number) {
    // both caller and callee can't already be connected
    no (caller + s.mapping.dialed) &
    (s.connected[Phone] + s.connected.Phone)
    ...
}

When re-ran the analysis, no counterexample found
Exercise 2: Hang Up Operation

Write an assertion to check that hangUp preserves the state invariant

```plaintext
pred hangUp[s, s' : Switch, caller : Phone] {
    // fill in the details here
    ...
}
```
Exercise 3: Call Blocking

Add call blocking functionality to the telephone system.

Each phone A has associated with a set of blocked numbers. A caller B cannot be granted a connection to A, if A has blocked B.
Exercise 4: Call Forwarding

Add call forwarding functionality to the telephone system.

Each phone A has associated with a forwarding number B. If C calls A, and A’s call forwarding is enabled, then C will be connected to B instead.
Exercise 5: Feature Interaction

Consider the following scenario.

A customer A has call forwarding enabled to number B, which has blocked C. The caller C requests a connection to A. What happens?

How does your Alloy model handle this?
Idioms
sig Name, Addr {}
sig Book {
    addr: Name -> lone Addr
}

pred add[b,b': Book, n: Name, a: Addr] {
    b'.addr = b.addr + n->a
}
Global State

- Book0 ($add_b'$)
  - addr [Name]
- Addr (a)
- Book1 (b)
- Name (n)
open util/ordering[Book]
sig Name, Addr {}
sig Book {
    addr: Name -> lone Addr
}
pred add[b,b': Book, n: Name, a: Addr] {
    b'.addr = b.addr + n->a
}
fact Traces {
    all b: Book - last |
    some n: Name, a: Addr | add[b, b.next, n, a]
}
Traces
Exercise 1

Implement the `del` operation using the traces idiom and visualize an addition and a deletion.
open util/ordering[Time]

sig Name, Addr, Time {}

sig Book {
    addr: Name -> lone Addr -> Time
}

pred add[b: Book, n: Name, a: Addr, t,t': Time] {
    b.addr.t' = b.addr.t + n->a
}
Local State
Exercise 2

Implement a predicate that adds one name to two address books simultaneously.
open util/ordering[Time] as ord

sig Name, Addr, Time {}

sig Book {
  addr: Name -> lone Addr -> Time
}

abstract sig Event {
  t, t': Time,
  book: Book
}

sig Add extends Event {
  n: Name,
  a: Addr }
{
  book.addr.t' = book.addr.t + n->a
}

fact Traces {
  all lt: Time - ord/last |
  some e: Event |
  e.t = lt && e.t' = lt.next
}
Events
Exercise 3

Add a del event to the event-based model and visualize an instance.
Checking Data Refinement

**Trace inclusion:** Does the concrete system conform to the abstract system?
module FileSystem [Data, FID]

sig File {
    contents : seq Data
}

sig AbsFileSys {
    fmap : FID -> lone File,
}
sig File {
    contents : seq Data
}
sig AbsFileSys {
    fmap : FID -> lone File,
}

fun readAbs [fsys : AbsFileSys, fid : FID] : seq Data {
    let f = fsys.fmap[fid] |
    f.contents
}
File Write Operation

pred \texttt{writeAbs} [\texttt{fsys, fsys}' : \texttt{AbsFileSys}, \texttt{fid} : \texttt{FID},
buffer : \texttt{seq Data}]
{ some f, f' : \texttt{File} |
  \texttt{frameConds}[\texttt{fsys, fsys}', f, f', \texttt{fid}] \text{ and }
  f'.\texttt{contents} = f.\texttt{contents} +++ buffer
}

pred \texttt{frameConds} [\texttt{fsys, fsys'} : \texttt{AbsFileSys}, f, f' : \texttt{File},
fid : \texttt{FID}]
{ f = \texttt{fsys.fmap}[\texttt{fid}]
  \texttt{fsys'.fmap} = \texttt{fsys.fmap} +++ (\texttt{fid} \to f')
}
Refining Abstract File System

[Diagram showing an abstract file and its concrete representation]
Concrete File System

module FileSystem [Data, FID]

sig File {
    blocks : seq Block,
    size : Int
}

sig Block { data : seq Data }
    #data = BLOCK_SIZE

fun BLOCK_SIZE : Int { 4 }

sig ConcFileSys { fmap : FID -> lone File }
Exercise 4: State Invariant

sig File {
    blocks : seq Block,
    size : Int
}

pred invariant [fsys: ConcFileSys] {
    // fill in the details here
    ...
}
Exercise 5: Read Operation

fun readConc [fsys: ConcFileSys, fid: FID] : seq Data {
  // fill in the details here
  ...
}

abstract file

0 1 0 0 1 1 1 0 0 0 1

concrete representation

0 1 0 0 1 1 1 0 0 0 1
fun readConc [fsys: ConcFileSys, fid: FID] : seq Data {
    let f = fsys.fmap[fid] |
    { i : Int, d : Data |
        let blockIdx = div[i, BLOCK_SIZE],
        block = f.blocks[blockIdx] |
        block.data[rem[i, BLOCK_SIZE]] = d and
        i < f.size }
}

Read Operation: A Solution
Abstraction Function

pred alpha [concFsys : ConcFileSys, absFsys : AbsFileSys] { 
    all fid : FID |
    let concFile = concFsys.fmap[fid],
        absFile = absFsys.fmap[fid] |
        #absFile.contents = concFile.size and
        (all idx : concFile.blocks.inds |
            let block = concFile.blocks[idx],
            dataFrag = block.data,
            from = mul[idx, BLOCK_SIZE],
            to = add[from, BLOCK_SIZE] - 1 |
            absFile.contents.subseq[from, to] = dataFrag
        )
}
Abstraction Function

```plaintext
pred alpha [concFsys : ConcFileSys, absFsys : AbsFileSys] {
    all fid : FID |
    let concFile = concFsys.fmap[fid],
        absFile = absFsys.fmap[fid] |
        #absFile.contents = concFile.size and
        (all idx : concFile.blocks.inds |
            let block = concFile.blocks[idx],
                dataFrag = block.data,
                from = mul[idx, BLOCK_SIZE],
                to = add[from, BLOCK_SIZE] - 1 |
                absFile.contents.subseq[from, to] = dataFrag
        )
}
```
Abstraction Function

pred alpha [concFsys : ConcFileSys, absFsys : AbsFileSys] {
    all fid : FID |
        let concFile = concFsys.fmap[fid],
        absFile = absFsys.fmap[fid] |
        #absFile.contents = concFile.size and
        (all idx : concFile.blocks.inds |
            let block = concFile.blocks[idx],
            dataFrag = block.data,
            from = mul[idx, BLOCK_SIZE],
            to = add[from, BLOCK_SIZE] − 1 |
            absFile.contents.subseq[from, to] = dataFrag
        )
    }
}
A function for abstraction:

\[
\text{pred } \alpha [\text{concFsys} : \text{ConcFileSys}, \text{absFsys} : \text{AbsFileSys}] \{ \\
    \text{all } \text{fid} : \text{FID} \mid \\
    \text{let concFile} = \text{concFsys.fmap}[\text{fid}], \\
    \text{absFile} = \text{absFsys.fmap}[\text{fid}] \mid \\
    \#\text{absFile.contents} = \text{concFile.size and} \\
    (\text{all } \text{idx} : \text{concFile.blocks.inds} \mid \\
    \text{let block} = \text{concFile.blocks}[\text{idx}], \\
    \text{dataFrag} = \text{block.data}, \\
    \text{from} = \text{mul}[\text{idx}, \text{BLOCK_SIZE}], \\
    \text{to} = \text{add}[\text{from}, \text{BLOCK_SIZE}] - 1 \mid \\
    \text{absFile.contents.subseq}[\text{from}, \text{to}] = \text{dataFrag} \\
    ) \\
\}
\]
Abstraction Function

pred alpha [concFsys : ConcFileSys, absFsys : AbsFileSys] { all fid : FID |
  let concFile = concFsys.fmap[fid],
  absFile = absFsys.fmap[fid] |
  #absFile.contents = concFile.size and
  (all idx : concFile.blocks.inds |
    let block = concFile.blocks[idx],
    dataFrag = block.data,
    from = mul[idx, BLOCK_SIZE],
    to = add[from, BLOCK_SIZE] - 1 |
    absFile.contents.subseq[from, to] = dataFrag
  )
}
Formulating Refinement Check

Traditionally:

\[ \forall c, c', a \mid C(c, c') \land \alpha(c, a) \quad \Rightarrow \quad \exists a' \mid \alpha(c', a') \land A(a, a') \]
Refinement Check with Model Finder

\[ \forall c, c', a \mid C(c, c') \land \alpha(c, a) \]
\[ \Rightarrow \exists a' \mid \alpha(c', a') \land A(a, a') \]

After negation & skolemization, unbounded universal quantifier appears:

\[ C(c, c') \land \alpha(c, a) \land \forall a' \mid \neg \alpha(c', a') \lor \neg A(a, a') \]

Finite model finder will generate a spurious counterexample!
Dealing with Unbounded ∀

If \( \alpha \) is **functional** & **total**, we can reformulate the refinement claim as:

\[
\forall c, c' \mid C(c, c') \Rightarrow A(\alpha(c), \alpha(c'))
\]
Checking Refinement in Alloy

```alloy
assert readOK {
    all concFsys : ConcFileSys, absFsys : AbsFileSys, fid : FID |
    invariant[concFsys] and alpha[concFsys, absFsys] =>
    readConc[concFsys, fid] = readAbs[absFsys, fid]
```
```
Exercise 5: Write Operation

Complete the write operation for the concrete file system, and check that it conforms to the abstract version.

```haskell
pred writeConc [fsys, fsys' : ConcFileSys, fid : FID, buffer : seq Data]
{
    // fill in the details here
    ...
}
```
Any Questions?

Eunsuk Kang (eskang@mit.edu)

Joe Near (jnear@mit.edu)