Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

Spring 2013

Lecture 3

Today's Topics

- Advanced Alloy language features (Ch. 3 and 4)
  - Cardinality Constraints
  - Integers
  - Modules and Polymorphism
- Advanced Alloy examples
  - Address book model (Ch. 2)
  - Memory model (Ch. 6)

Cardinality Constraints

- The cardinality operator # applied to a relation gives the size of the relation
- Examples:

  name = {(N_0),(N_1),(N_2)}

  address = {(N_0,A_0), (G_0,A_0), (G_0,A_2), (N_1, A_1)}

  #name = 3 #address = 4

  Groups contain at least two elements:

```
all g : Group | #g.address > 1
```

Cardinality Operator: Exercise

- Does the following constraint hold for all sets `A: set univ` and `B: set univ`?

```
# (A + B) = #A + #B
```

- How can one use the cardinality operator to express that the function `f: univ->univ` is bijective?

Integers

- Alloy supports integers and they can be embedded into relations of type `Int`
  - `Int[i]` converts integer `i` to `Int` atom
  - `int[a]` retrieves embedded integer from `Int` atom `a`

- Example: a graph with weighted edges and a constraint that self-edges have zero weight

```
sig Node { adj: Node ->lone Int }
fact {
  all n: Node |
    let w = n.adj[n] | some w => int[w]=0
}
```

Integer Operators

`a + b`	`a.add[b]`	addition
`a - b`	`a.sub[b]`	subtraction
	`a.mul[b]`	multiplication
	`a.div[b]`	division
	`a.rem[b]`	remainder
`- a`		negation
`a = b`		equal
`a != b`		not equal
`a < b`		less than
`a > b`		greater than
`a <= b`		less than or equal to
`a >= b`		greater than or equal to
`a << b`		left-shift
`a >> b`		sign-extended right-shift
`a >>> b`		zero-extended right-shift

`add`, `sub`, `mul`, `div`, and `rem` require

```
open util/integers
```

Sum Expressions

- `sum x: e | ie`
  - denotes the integer obtained by summing the values of the integer expression `ie` for all values of the scalar `x` drawn from the set `e`.
- Example: The size of a group is the sum of the sizes of its subgroups

```
address: Group->Addr   //maps groups to their addresses
split: Group->Group  //partitions groups into disjoint subgroups
all g: split.Group |
   #g.address = (sum g': g.split | #g'.address)
```

Modules and Polymorphism

Alloy has a simple module system that allows to split a model among several modules.

- The first line of every module is a module header

  `module` *`modulePathName`*

- Every module that is used must be explicitly imported using an **open** statement

  `open` *`modulePathName`*

- Modules can extend signatures that are declared in imported modules.

Modules: Example

A module that defines a predicate true of relations that are acyclic

```
module library/graphs
pred Acyclic [r: univ->univ] {no ^r & iden}
```

and a use of this module in a model of a family

```
module family
open library/graphs
sig Person {parents: set Person}
fact {Acyclic [parents]}
```

Modules and Namespaces

- Modules have their own namespaces.
- Name clashes can be resolved by referring to components with qualified names.
- A signature, predicate, or function X in a module of path name p has qualified name p/X
- Example:

```
module family
open library/graphs
sig Person {parents: set Person}
pred Acyclic []
  {library/graphs/Acyclic [parents]}
```

Namespace Aliases

- Module path names can be abbreviated using aliases.

- Example:

```
module family
open library/graphs as g
sig Person {parents: set Person}
pred Acyclic [] {g/Acyclic [parents]}
```

Parametric Modules

- A module can be parameterized by one or more signature parameters.
- Example: A parameterized module

```
module library/graphs [t]
pred Acyclic [r: t->t] {no ^r & iden}
```

and a module that uses it

```
module family
open library/graphs [Person]
sig Person {parents: set Person
fact {Acyclic [parents]}
```

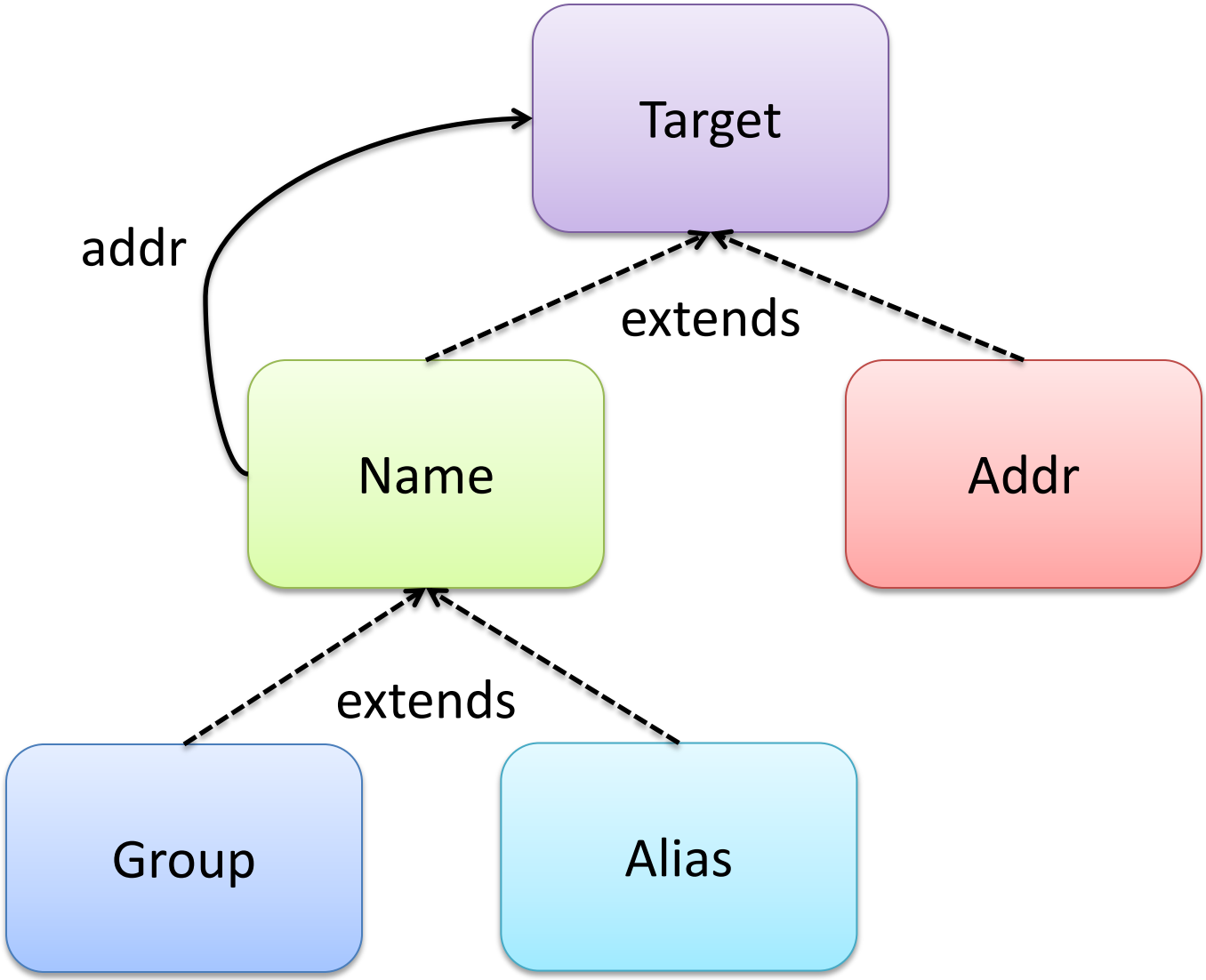
Predefined Modules

- Alloy comes with many predefined modules that are useful when developing your own models.

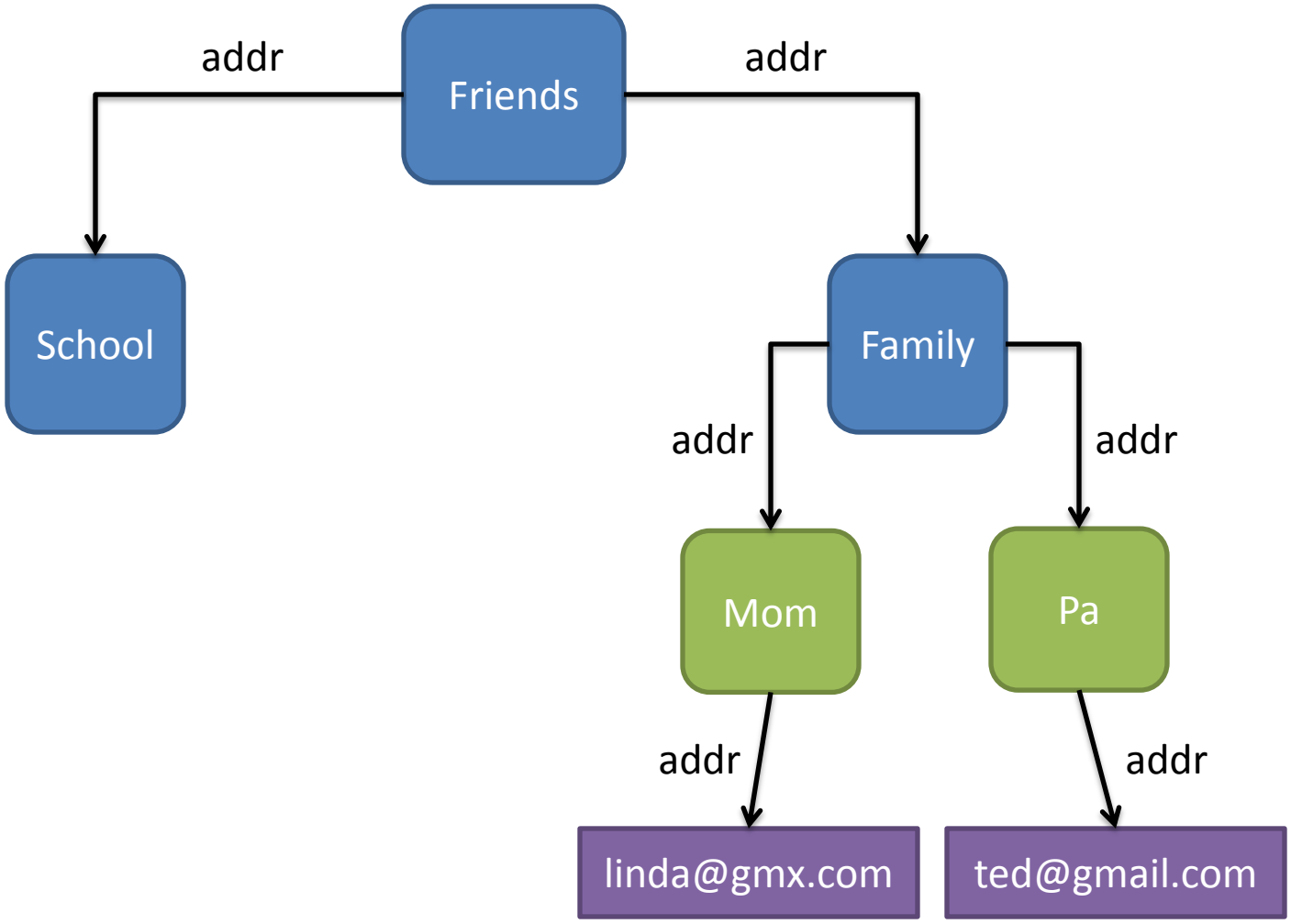
- See, e.g., the modules in "util" when using "Open Sample Models…"

Advanced Examples

Hierarchical Address Book



How can we avoid empty groups?



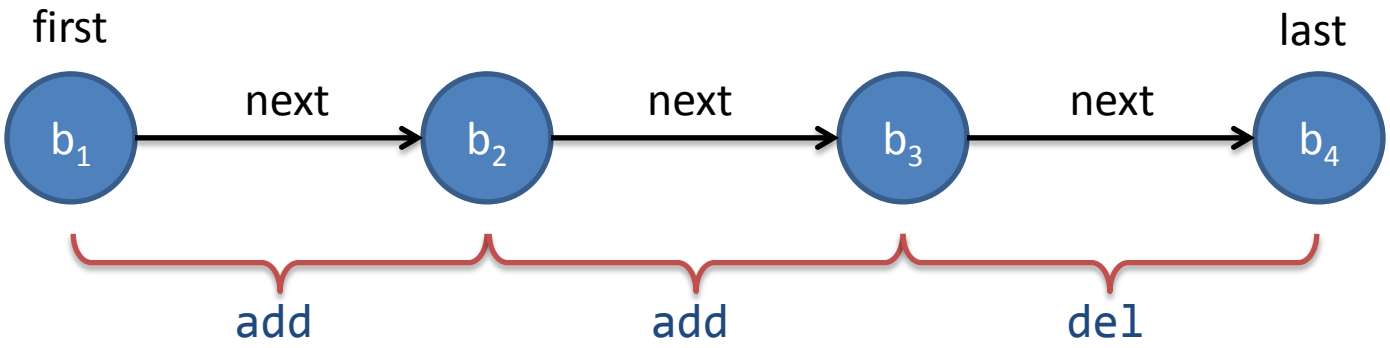
Invariant Properties

- The non-emptiness property does not follow from the facts of our address book model.
- We do not want to assume a priori that all groups of the address book are non-empty.
- Rather, we want this property to be invariant under all address book operations.

Invariants and Traces

- A trace of a model is the consecutive sequence of states obtained by the execution of a sequence of operations of a model starting from an initial state.
- An invariant of a model is a property that holds for all states of all traces of a model.

Address Book Traces



Address book operations define a binary transition relation on states of the address book.

The transition relation defines the set of traces.

Summary

- We can use Alloy to
  - model traces using transition relations of operations and
  - formulate invariants as assertions about traces.
  - Counterexamples to such assertions are traces showing how the invariant is violated.
- Often we have to restrict operations by adding appropriate preconditions to ensure that all invariants are preserved.

Operations: Events vs. Predicates

- Sometimes it is more convenient to model operations as events that are atoms in the model rather than as predicates.
- This makes it easier to identify which operations fired in a counterexample trace.

Event-based Address Book Model

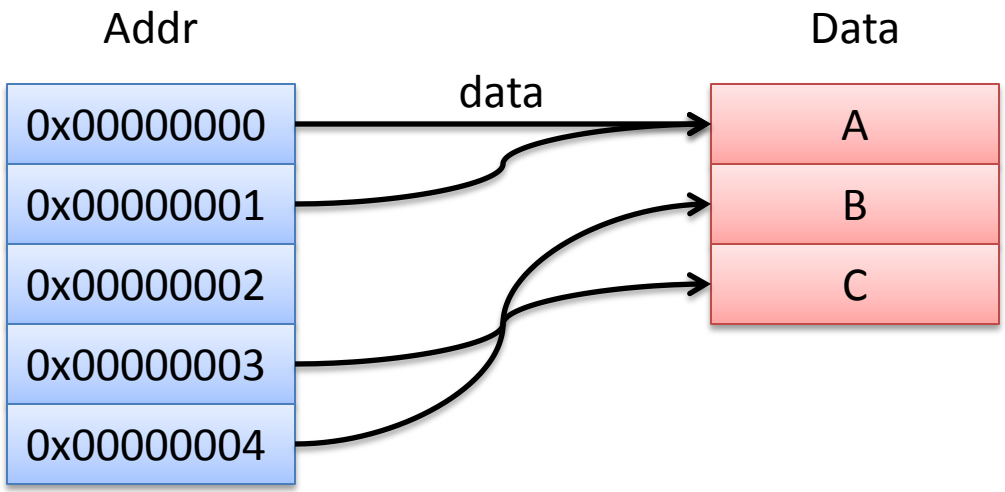
```
abstract sig Event {
  pre, post: Book,
  n: Name, t: Target
}

sig AddEvent extends Event {} {
  t in Addr or some lookup [pre, t]
  post.addr = pre.addr + n->t
}
```

Example: Memory Model

Next, we will learn

- how to use non-determinism in specifications;
- how to relate models at different levels of abstractions;
- how to effectively use the module system.

Abstract Memory Model

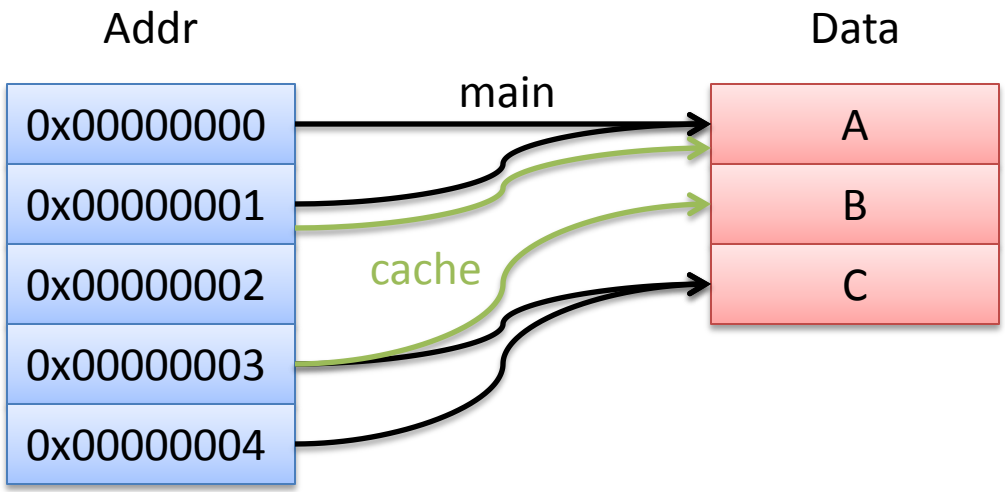


A memory is a partial function between addresses and data.

Abstract Memory: Operations

- `write [m, m': AbsMemory, a: Addr, d: Data]`
  - overwrite mapping of address `a` in `m` to `d`
  - resulting memory is `m'`
- `read [m: AbsMemory, a: Addr, d: Data]`
  - read data entry associated with address `a` in `m`
  - result of read is `d`
  - state of memory `m` remains unchanged
  - if `m` has no entry for `a`, result of read can be arbitrary

Cache Memory Model



Cache system consists of main memory and cache memory.

Cache Memory: Operations

- `write [c, c': Cachememory, a: Addr, d: Data]`
  - overwrite mapping of address `a` in cache of `c` to `d`
  - resulting memory is `c'`
  - state of main memory of `c` remains unchanged

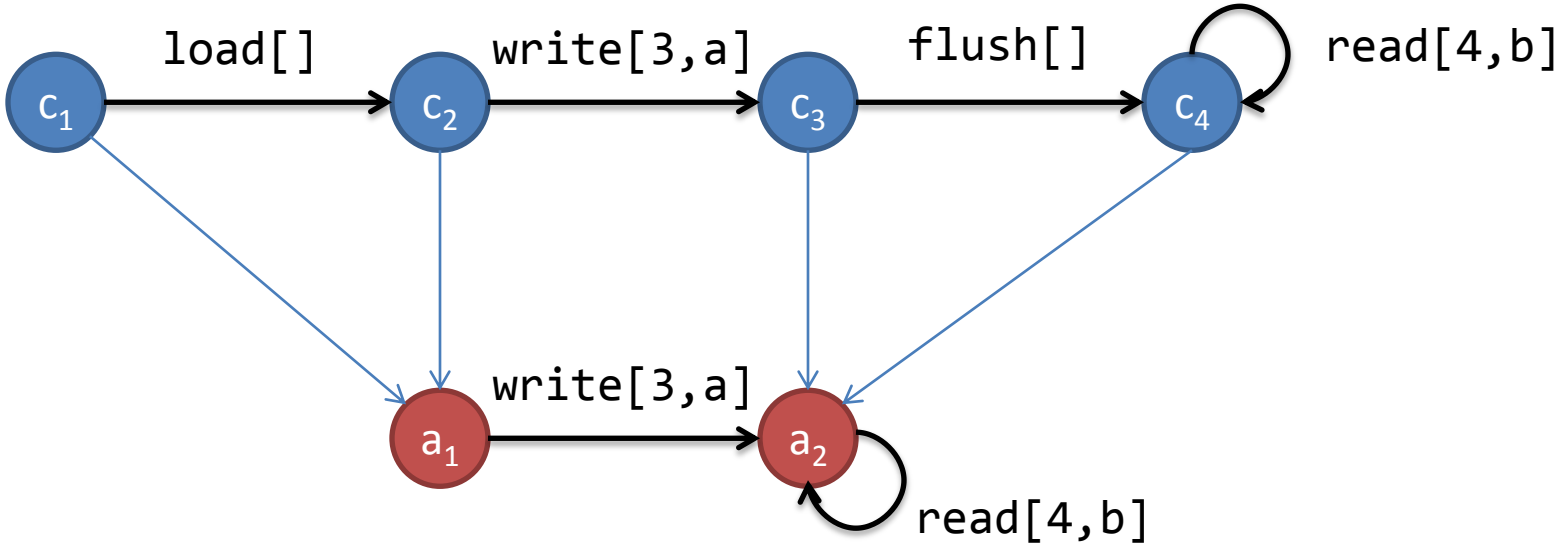
- `read [c: CacheMemory, a: Addr, d: Data]`
  - read data entry associated with address `a` in cache of `c`
  - result of read is `d`
  - state of memory `c` remains unchanged
  - precondition: some entry for `a` exists in the cache of `c`

Cache Memory: Operations (contd.)

- `load [c, c': CacheMemory]`
  - non-deterministically load some data values from the main memory of `c` to the cache of `c`
  - resulting memory is `c'`
  - caching policy is unspecified
- `flush [c, c': CacheMemory]`
  - non-deterministically flush some data values from the cash of `c` to the main memory of `c`
  - resulting memory is `c'`
  - caching policy is unspecified

Abstraction and Refinement

It appears that for every trace of the cache memory we can find a corresponding trace of the abstract memory.



To prove this property, we have to formally relate cache memories and abstract memories.

Abstraction Functions [Hoare, 1972]

An abstraction function maps concrete states to corresponding abstract states.

Suppose our operations are

```
pred concreteOp [s, s': ConcreteState] {…}
pred abstractOp [s, s': AbstractState] {…}
```

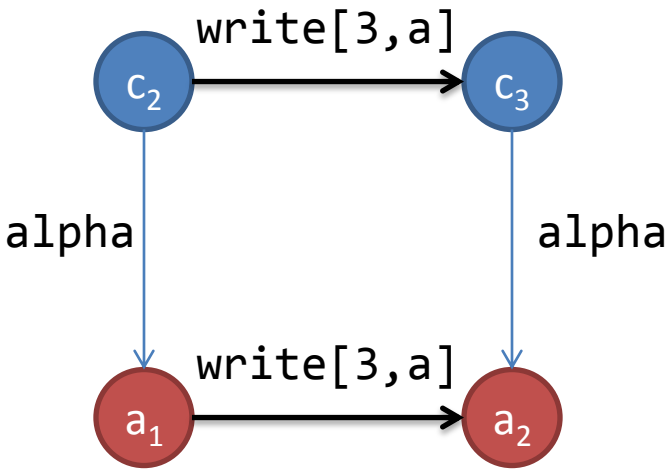
Then we try to find an abstraction function

```
fun alpha [s: ConcreteState]: AbstractState {…}
```

that makes this assertion valid:

```
assert AbstractionRefinement {
  all s, s': ConcreteState |
   concreteOp [s, s'] =>
     abstractOp [alpha[s], alpha[s']]
}
```

Abstraction Functions



For every concrete transition there is a corresponding abstract transition labeled by the same event.

Abstraction Function for Cache Memory

```
module memory/checkCache [Addr, Data]
open cacheMemory [Addr, Data] as cache
open abstractMemory [Addr, Data] as amemory

fun alpha [c: CacheMemory]: AbsMemory {
  {m: AbsMemory | m.data = c.main ++ c.cache}
}

assert WriteOK {
  all c,c': CacheMemory, a:Addr, d:Data, m,m': AbsMemory |
    cache/write [c, c' a, d]
    and m = alpha [c] and m' = alpha [c']
      => amemory/write [m, m', a, d]
}
```

Abstraction Functions and Conformance

- A concrete model C conforms to an abstract model A if the set of event traces of C is a subset of the set of event traces of A.
  - properties that hold for all event traces of A also hold for all traces of C.
- Abstraction functions are sound…
  - If an abstraction function between C and A exists, then C conforms to A.
- …but not complete.
  - If the non-determinism in the concrete and abstract models are observed at different points in time then an abstraction function might not exist.

Soundness of Abstraction

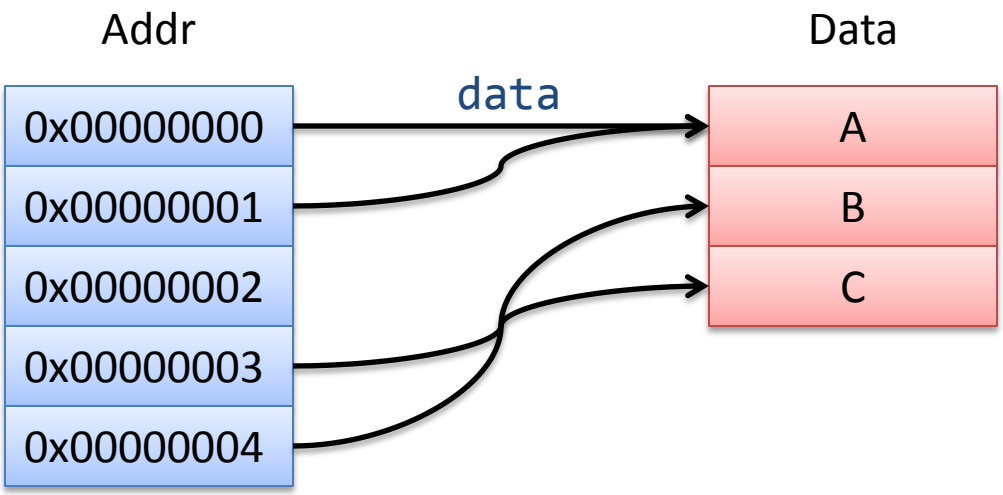
If `abstractMemory` conforms to `cachedMemory` then

```
all m,m': AbsMemory, a: Addr, d1,d2: Data |
  write [m, m', a, d1] and read [m', a, d2]
      => d1 = d2
}
```

implies

```
all c,c': CacheMemory, a: Addr, d1,d2: Data |
  write [c, c', a, d1] and read [c', a, d2]
      => d1 = d2
}
```

Fixed Size Memory Model

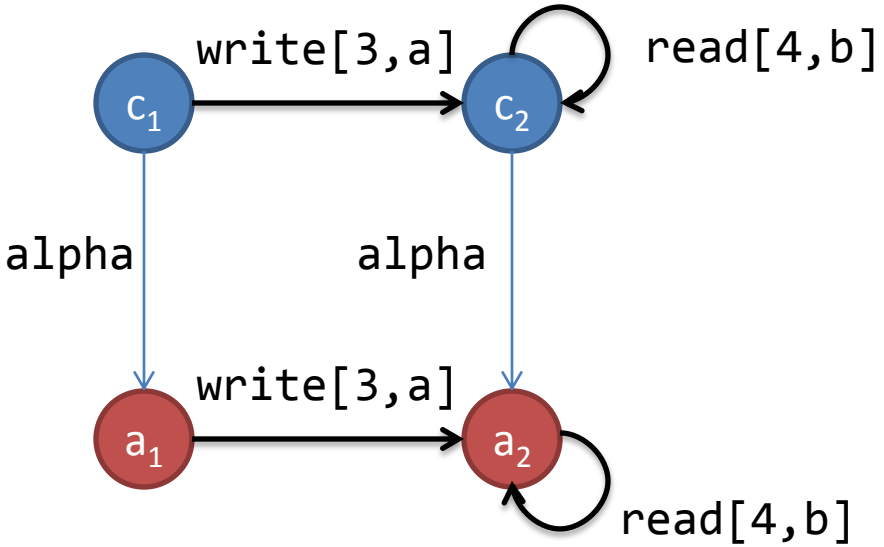


Initially, all addresses have some arbitrary data value associated, i.e., `data` is total.

The only non-determinism is in the choice of the initial state. Then all operations are deterministic.

Abstraction Function for Fixed Size Memory

The fixes size memory should still conform to the abstract memory.



But there does not seem to be an abstraction function: How do we distinguish written values from the arbitrary initial values in concrete memories?

History and Prophecy Variables

- History Variables:
  - Augment concrete states with an auxiliary variable `unwritten` that keeps track of the addresses not yet written after initialization.
  - `unwritten` is called a history variable, because it holds additional history about the behavior.
- Prophecy Variables:
  - the dual concept to a history variable is a prophecy variable; it holds information about the future behavior of a state.
- Using history and prophecy variables one can always find an appropriate abstraction function to prove conformance.

Adding a History Variable

```
sig FixMemory_H extends FixMemory {
  unwritten: set Addr
}

pred init [m: FixMemory_H] {
  memory/init [m]
  m.unwritten = Addr
}

pred write [m, m': FixMemory_H, a: Addr, d: Data] {
  memory/write [m, m', a, d]
  m'.unwritten = m.unwritten - a
}
```

Abstraction Function for Fixed Size Memory

```
module memory/checkFixedSize [Addr, Data]
open fixedSizeMemory_H [Addr, Data] as fmemory
open abstractMemory [Addr, Data] as amemory

pred alpha [fm: FixMemory_H, am: AbsMemory] {
  am.data = fm.data - (fm.unwritten->Data)
}

assert InitOK {
  all fm: FixMemory_H, am: AbsMemory |
    fmemory/init [fm] and alpha [fm, am]
      => amemory/init [am]
}
```

Summary

- We can relate models at different levels of abstraction using abstraction functions.
- Abstract models have more behavior than the concrete models that refine them and are typically non-deterministic.
- Abstraction functions allow us to efficiently check complex properties of concrete models on simpler abstract models.