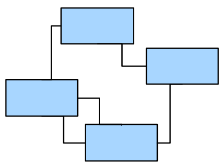
Modelling with Alloy

Adapted from from

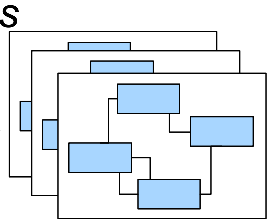
Greg Dennis and Rob Seater
Software Design Group, MIT

Static vs. dynamic models

- Static model
  - – describes states, not behaviors
  - – properties are *invariants*
  - – *e.g. that a list is sorted*



- *Dynamic model*
  - – *describe transitions between states*
  - – *properties are operations*
  - – *e.g. how a sorting algorithm works*



Modeling entities

- University course catalog and requirements for choosing courses
  - First step of building a model
    - consider what things are relevant
    - structure them hierarchically
    - subsets for orthogonal classification

    our system has *courses*, *students, curricula (e.g., CS or EE)*

    all courses are offered in certain semesters and have certain prerequisites

    courses can be for graduate or undergraduate students

    - why *not* include in your classification . . . ?
      - » instructor
      - » rooms where courses meet

Modeling relationships

- ➢ Create fields for the following
  - course belongs to a single curriculum (e.g., CS)
  - CS has courses required to graduate
  - a course has one or more prerequisites
  - students have their course plan
  - student has at most one *major* curriculum from which courses can be selected
  - can also take max number of courses from others

Hints

- We'll give hints on how to specify
- Hints sometimes similar to "specification patterns"

Specification hint: definition

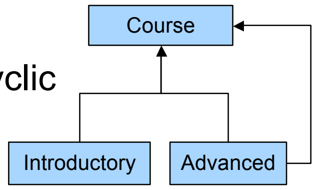
- Define a new term using existing terms
  - declare new relation and constrain to existing relations
  - constraint often written as equality, e.g.

```
sig Person {
  spouse: lone Person,
  parents: set Person,
  inlaws: set Person
}
fact { inlaws = spouse.parents }
```

➢ define a term for curricula, courses, required or elective, ….

Specification hint: composite

- Prerequisites establish *composite* hierarchy
  - advanced courses are composites
  - introductory courses are leafs
  - another example: file system directories and files



- composites typically must be acyclic
  - e.g. directory cannot contain itself

➢ constrain prerequisite relation to be acyclic
  - course cannot be its own prerequisite

Specification hint: sanity check

- Write simple assertions while building models

- You'll be surprised how many fail

➢ check that every advanced course has an introductory course that precedes it

Functions and predicates

➢ Create predicates or functions for the following

- condition that a student can take a course
  - student has taken prereqs but not course itself
- for a set of courses, expression for complete prereqs
  - prereqs of prereqs, prereqs of prereqs of prereqs, etc
- condition that a student can graduate
  - has taken all course's required by dept
  - …

Specification hint: guided simulation

- Simulates model to check consistency
  - does the model admit any instances?
  - explore typical & interesting configurations

➢ create predicates with desired configurations
  - run predicates to ensure they exist

- example configuration:
  - a student's plan with at least one advanced course
  - at least one student can graduate

Specification hint: multirelation

- use higher-arity relation to model relationship between more than two entities
- address book example:

```
sig Book {
  addrs: Name -> Addr
}
```

➢ create a set of grades

➢ student has a grade in each course taken

Specification hint: singleton

- Particular elements of set play important roles
- Use **one** multiplicity to make a singleton sig

```
one sig Root extends Directory {}
```

➢ introduce courses taken in other Universities (Erasmus, …) where grades are exactly A, B, C, D, and F

➢ update passing condition so student must get C or better in each course taken

Specification hint: approximation

- Omit/loosen constraints present in reality
  - don't need to model everything!
- Looser model often good enough
  - if abstraction, property preservation is sound
- Important to keep approximations in mind!!!

Specification hint: check and visualize

- Write assertion that if a student can graduate, they must have passed all required courses as well as transitive prerequisites of required courses
- Check assertion and visualize it
- Add sensible constraints to ensure assertion passes

Specification hint: set object

- All relations in Alloy are first order
- *but* some relationships are higher-order
  - relate sets of elements, not individuals

- Solution: represent sets themselves as objects
  - single field relating set to its elements
  - often canonicalized: no two sets have same elements

➢ allow curricula multiple sets of required courses
  - student can fulfill anyone of those sets

Dynamic modeling

Address book revisited

Model of an address book

```
abstract sig Target {}
sig Name extends Target {}
sig Addr extends Target {}

sig Book { addr: Name -> Target }

pred init [b: Book] { no b.addr }

pred inv [b: Book] {
  let addr = b.addr | all n: Name {
    n not in n.^addr
    some addr.n => some n.addr
  }
}

fun lookup [b: Book, n: Name] : set Addr {
  n.^(b.addr) & Addr
}

assert namesResolve {
  all b: Book | inv[b] =>
    all n: Name | some b.addr[n] => some lookup[b, n]
}
check namesResolve for 4
```

What about operations?

- How is a name & address added to/deleted from a book?
- No built-in model of execution
  - no notion of time or mutable state
- Need to model time/state explicitly
- Can use a new “book” after each mutation:

```
pred add [b, b': Book, n: Name, t: Target] {
  b'.addr = b.addr + n->t
}
```

Testing operations

- We can simulate effect of an operation, e.g., add

➢ by creating both interesting valid and invalid states

```
pred showAdd [b, b': Book, n: Name, t: Target] {
  valid[b] //or invalid
  add[b, b', n, t]
}
```

Specification hint: abstract machine

- Treat actions as operations on global state

```
sig State {…}

pred init [s: State] {…}

pred inv [s: State] {…}

pred op1  [s, s’: State] {…}
…
pred opN  [s, s’: State] {…}
```

- in addressBook, *State* is *Book*
  - each *Book* represents a new system state

Specification hint: invariant preservation

- Check that an operation preserves an invariant

```
assert initEstablishes {
  all s: State | init[s] => inv[s]
}
check initEstablishes

// for each operation
assert opPreserves {
  all s, s': State |
    inv[s] && op[s, s'] => inv[s']
}
check opPreserves
```

➢ apply this pattern to the addressBook model
➢ do the *add* and *delete* ops preserve the invariant?

Specification hint: operation preconditions

- Include precondition constraints in an operation
  – operations no longer total
- the *add* operation with a precondition:

```
pred add[b, b': Book, n: Name, t: Target] {
  // precondition
  t in Name => (n !in t.*(b.addr) && some b.addr[t])
  // postcondition
  b'.addr = b.addr + n->t
}
```

➢ check that *add* now preserves the invariant
➢ add a sensible precondition to the delete operation
  – check that it now preserves the invariant

What about traces?

- We can check properties of individual transitions and properties of sequences of transitions to obtain entire system simulation of a sequence of operations

➢ algorithm correctness
  – check that all traces end in a desired final state

➢ planning problems
  – find a trace that ends in a desired final state

Specification hint: traces

- Traces model sequences of executions of abstract machine
  - Create linear (total) ordering over states
  - Connect successive states by operations
    - constrains all states to be reachable

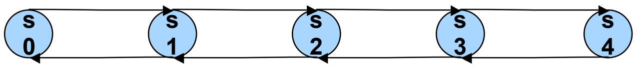
```
open util/ordering[State] as ord
…
fact traces {
  init [ord/first]
  all s: State - ord/last |
    let s' = s.next |
      op1[s, s'] or … or opN[s, s']
}
```

Ordering module

- Establishes linear ordering over atoms of signature *S*

```
open util/ordering[S]
```

`S = s0 + s1 + s2 + s3 + s4`




```
first = s0
last = s4
s2.next = s3
s2.prev = s1
s2.nexts = s3 + s4
s2.prevs = s0 + s1
```

```
lt[s1, s2] = true
lt[s1, s1] = false
gt[s1, s2] = false
lte[s0, s3] = true
lte[s0, s0] = true
gte[s2, s4] = false
```

Specification hint: safety properties

- Can check safety property with one assertion
  - because all states are reachable

```
pred safe[s: State] {…}

assert allReachableSafe {
  all s: State | safe[s]
}
```

Static vs dynamic models

- Static traffic light model

```
sig Color {}
sig Light {
  color: Color
}
```

- Dynamic traffic light model with abstract machine
  – all dynamic components collected in one sig

```
sig Color {}
sig Light {}
sig State {
  color: Light -> one Color
}
```

Specification hint: local state

- Embed state in individual objects
  – variant of abstract machine
- Move state/time signature out of first column
  – typically most convenient in last column

global state

```
sig Color {}

sig Light {}

sig State {
  color: Light -> one Color
}
```

local state

```
sig Time {}

sig Color {}

sig Light {
  color: Color one -> Time
}
```