Verifying Safety & Liveness in Alloy

Alcino Cunha

November 3, 2014

Safety properties

- *Something bad will not happen*!
- A property ϕ is a safety property if it must be true at all reachable states.
- A counter-example to a safety property is a finite prefix of a path that leads to a state where ϕ does not hold.

Liveness properties

- *Something good will happen*!
- A property ϕ is a liveness property if it must eventually be true at some state in all paths starting from all initial states.
- A counter-example to a liveness property is an (infinite) path where ϕ never holds.

The farmer puzzle: static model

```
abstract sig Being {
  eats : set Being,
  where : one Bank
}
one sig Farmer, Wolf, Sheep, Beans extends Being {}

fact Eats {
  eats = Farmer->Being + Sheep->Beans + Wolf->Sheep
}

abstract sig Bank {
  cross : one Bank
}
one sig Left, Right extends Bank {}

fact Cross {
  cross = Left->Right + Right->Left
}
```

Modeling state machines in Alloy

- Add a signature `State` for representing states.
- Add `State` as an extra column to all mutable relations.
  - *Global-state idiom*: the `State` is the first column - all mutable relations are declared in the `State` signature.
  - *Local-state idiom*: the `State` is the last column - each mutable relation is still declared in the same signature as before.
- Specify the initial states with a predicate.
- Specify transitions (operations) using predicates relating pre- and post-states (with pre- and post-conditions).
- Do not forget the *frame-conditions* to specify what is unchanged!

The farmer puzzle: dynamic model and initial state

```
sig State {}

abstract sig Being {
  eats : set Being,
  where : Bank one -> State
}

...

pred init [s : State] {
  Being = (s.where).Left
}
```

The farmer puzzle: crossing alone

```
pred alone [s,s' : State] {
  // Pre-conditions
  no x,y : (s.where).(Farmer.(s.where))-Farmer | x in y.eats

  // Post-conditions
  Farmer.(s'.where) = Farmer.(s.where).cross

  // Frame-conditions
  all b : Being-Farmer | b.(s'.where) = b.(s.where)
}
```

The farmer puzzle: crossing with another being

```
pred notalone [b : Being, s,s' : State] {
  // Pre-conditions
  b != Farmer
  b.(s.where) = Farmer.(s.where)
  no x,y : (s.where).(Farmer.(s.where))-(Farmer+b) | x in y.eats

  // Post-conditions
  Farmer.(s'.where) = Farmer.(s.where).cross
  b.(s'.where) = b.(s.where).cross

  // Frame-conditions
  all x : Being-(Farmer+b) | x.(s'.where) = x.(s.where)
}
```

The farmer puzzle: some properties

- Safety properties:
  - The beings never eat each other.
  - The beings will never be together in the right margin (if not true, a counter-example solves the puzzle).
- Liveness properties:
  - The beings will always end up together in the right margin.

The farmer puzzle: some properties

```
pred noeating [s : State] {
  all b : Bank {
    Farmer.(s.where) = b
    or
    no x,y : (s.where).b | x in y.eats
  }
}

pred notright [s : State] {
  Being not in (s.where).Right
}

pred allright [s : State] {
  Being in (s.where).Right
}
```

Verification with the indirect (or inductive) method

- For safety property ϕ:
  - Check that ϕ holds in the initial states.
  - Check that ϕ is preserved by all operations.
- For liveness property ϕ:
  - Find a postive metric on states that is zero iff ϕ holds.
  - Check that it strictly decreases with all operations.
- This method over-approximates the set of reachable states, and is geared towards verification:
  - If the above checks hold the property is true.
  - If not, verification is inconclusive (counter-examples may be invalid).

Verifying `noeating` with the inductive method

```
check init_satisfies_noeating {
  all s : State |
    init[s] implies noeating[s]
} for 3 but 1 State

check alone_preserves_noeating {
  all s,s' : State |
    noeating[s] and alone[s,s'] implies noeating[s']
} for 3 but 2 State

check notalone_preserves_noeating {
  all s,s' : State, b : Being |
    noeating[s] and notalone[b,s,s'] implies noeating[s']
} for 3 but 2 State
```

Verification with the direct method

- Model valid path prefixes over the state machine.
  - A popular idiom to do so in Alloy is to use the `util/ordering` module, and represent prefixes with a total order on states.
- For safety property ϕ:
  - Check that ϕ holds for all states in all path prefixes.
- For liveness property ϕ:
  - Check that ϕ holds in some state in all paths prefixes with a back loop (i.e. modeling infinite paths).
- This method under-approximates the set of reachable states, and is geared towards falsification:
  - If a counter-example is found the property is false.
  - If not, verification is inconclusive (a longer prefixe might reach a problematic state).

Refuting `notright` with the direct method

```
open util/ordering[State]
...
fact valid_path_prefixes {
  init[first]
  all s : State - last {
    alone[s,s.next]
    or
    some b : Being | notalone[b,s,s.next]
  }
}

// The following check yields a counter-example that
// is the solution to the puzzle.

check puzzle_cannot_be_solved {
  all s : State | notright[s]
} for 3 but 8 State
```

Refuting `allright` with the direct method

```
// Two states are equal if all mutable relations are equal.

pred equal [s,s' : State] {
  s.where = s'.where
}

// A path prefix has a loop if two states are equal.

pred loop {
  some disj s,s' : State | equal[s,s']
}

// The following check yields a counter-example where
// the farmer keeps crossing the sheep forward and backward.

check puzzle_will_always_be_solved {
  loop implies (some s : State | allright[s])
} for 3 but 3 State
```

Homework

- What are the (ideally, weakest) pre-conditions that must be added to the operations so that `allright` holds?
- After adding such pre-conditions, can you find a metric to verify `allright` with the direct method?