MFES - Métodos Formais em Engenharia de Software

Alloy

Ana Paiva
apaiva@fe.up.pt www.fe.up.pt/~apaiva

Seq

- A new reserved keyword "seq" has been added for declaring a field as a sequence of atoms. In the following example, for each person p, "p.books" is a sequence of Book:

```
sig Book { }
sig Person { books: seq Book }
```

- The actual type of a sequence of Book is "Int->Book".
- So, if s is a sequence of Book, then the first element is s[0] and you can get the set of all elements by writing "univ.s"
- You can also use "seq" in quantifications and in function argument declaration, like this:

```
some s: seq Book | FORMULA
fun getAllElements [s: seq Book] : set Book { univ.s }
```


Seq

- Just like the other multiplicity symbols, when you use “seq” in a function argument declaration, we do not enforce that you always call the function/predicate with a well-formed sequence. So it is only for documentation purpose, to denote s is a binary relation from Int->Book.
- Note: for efficiency, we bound the length of allowed sequences. You can change this bound by setting the scope on “seq”. For example, if you want to allow sequences of up to 4 elements, you write

  check SomeAssertion for 4 seq

Seq

- To make it easier to manipulate sequences, we provide a number of helper functions: (these are defined in the pre-included util/sequniv.als file)
- **#s** - Return the number of elements in sequence s.
- **s.elems -** Return the set of elements in sequence s.
- **s.first -** If #s > 0, it returns the first element of s Otherwise, it returns the empty set
- **s.last -** If #s > 0, it returns the last element of s Otherwise, it returns the empty set
- **s.rest -** If #s > 1, it returns s with its first element removed Otherwise, it returns the empty sequence

Seq

- **s.butLast -** If #s > 1, it returns s with its last element removed Otherwise, it returns the empty sequence
- **s.isEmpty -** It returns true if #s==0.
- **s.hasDups -** It returns true if s contains duplicate elements.
- **s.inds -** If #s > 0, it returns the set of integers {0 .. (#s)-1} Otherwise, it returns the empty set
- **s.lastIdx -** If #s > 0, it returns the integer (#s)-1 Otherwise, it returns the empty set
- **s.afterLastIdx -** If (#s < the longest allowed sequence length), it returns #s. Otherwise, it returns the empty set

Seq

- **s.idxOf [x] -** If x does not appear in s, it returns the empty set. Otherwise, it returns the first index where x appears in s.
- **s.lastIdxOf [x] -** If x does not appear in s, it returns the empty set. Otherwise, it returns the last index where x appears in s.
- **s.indsOf [x] -** If x does not appear in s, it returns the empty set. Otherwise, it returns the set of indices where x appears in s.
- **s.add [x] -** If (#s < the longest allowed sequence length), it appends x to s. Otherwise, it returns s.
- **s.setAt [i, x] -** Precondition: 0 <= i < #s. It returns a new sequence where the i-th entry is changed to x.

Seq

- **s.insert [i, x]** - Precondition: 0 <= i <= #s
  It returns a new sequence where x is inserted at index i.
  Note: if #s was already equal to the longest allowed sequence length, then the last element of s will be removed first.
- **s.delete [i] -** Precondition: 0 <= i < #s
  It returns the result of deleting the element at index i
- **a.append [b] -** Returns the result of concatenating sequence a and sequence b (If the resulting sequence is too longer, it will be truncated)
- **s.subseq [from, to] -** Precondition: 0 <= from <= to < #s
  Returns the subsequence between from and to, inclusively.

References

- More details about Alloy Analyzer Visualization in
- https://cs.uwaterloo.ca/sites/ca.computer-science/files/uploads/files/CS-2013-04.pdf

Exemplo: Leader election in a ring

- A distributed algorithm for leader election (the process that coordinates the others);
- We'll consider the case in which the processes form a ring.
- We'll assume that processes have unique identifiers that are totally ordered;
- A simple and well-known protocol has the processes pass their identifiers as Tokens around the ring in some direction (say clockwise). Each process examines each identifier it receives. If the identifier is less than its own identifier, it consumes the token. If the identifier is greater than its own, it passes the token on. If the identifier equals its own identifier, it knows the token must have passed all the way around the ring, so it elects itself leader.

Exemplo - *ring election* (1)

```
module book/ringElection1
open util/ordering[Time] as TO
open util/ordering[Process] as PO

sig Time {}

sig Process {
    succ: Process,
    toSend: Process -> Time,
    elected: set Time
}

fact ring {
    all p: Process | Process in p.^succ
}

pred init [t: Time] {
    all p: Process | p.toSend.t = p
}
pred step [t, t': Time, p: Process] {
  let from = p.toSend, to = p.succ.toSend |
    some id: from.t {
      from.t' = from.t - id
      to.t' = to.t + (id - p.succ.prevs)
    }
```

The processes are to form a ring. The declaration of *succ* ensures that each process has exactly one successor, so all we need to add is the constraint that all processes are reachable from any process by following *succ* repeatedly:

The protocol itself is described in three stages. First, we record the initial condition—that each process is ready to send only its own identifier:

Second, we describe the allowed state transitions. In a given step, an arbitrary identifier (*id*) is chosen from the pool associated with a process (*from*) and moved to the pool associated with its successor (*to*):

Exemplo - *ring election* (2)

```
fact defineElected {
  no elected.first
    all t: Time-first |
      elected.t = {p: Process | p in p.toSend.t - p.toSend.(t.p
}

fact traces {
  init [first]
    all t: Time-last |
      let t' = t.next |
        all p: Process |
          step [t, t', p] or step [t, t', succ.p] or skip [t, t', p]
}

pred skip [t, t': Time, p: Process] {
  p.toSend.t = p.toSend.t'
}

pred show { some elected }
run show for 3 Process, 4 Time
// This generates an instance

assert AtMostOneElected { lone elected.Time }
check AtMostOneElected for 3 Process, 7 Time
// This should not find any counterexample

assert AtLeastOneElected { some t: Time | some elect
check AtLeastOneElected for 3 Process, 7 Time
// This generates a counterexample in which nothing happens
```

Third, we describe the designation of elected processes. At the first moment in time, no processes are elected; at other times, the set of processes elected is the set of processes that just received their own identifiers:

Here is the trace constraint:

It says that the initial condition holds for the first moment in time, and that for any subsequent time, each process *p* either takes a step, or its predecessor *succ.p* takes a step, or it does nothing. Doing nothing is modeled as an operation:

It's good to start with a simple simulation, to check that the model isn't overconstrained. For example, we might ask to see an execution in which some process gets elected:

Here is an assertion claiming that there is at most one elected process:

This second assertion is invalid; it has a counterexample in which nothing happens at all. The problem is including the *skip* operation, which allows every process to skip in every step!

Exemplo - *ring election* (3)

```
pred progress  {
  all t: Time - TO/last |
    let t' = TO/next [t] |
    some Process.toSend.t => some p: Process | not skip [t, t', p]
}

assert AtLeastOneElected { progress => some elected.Time }
check AtLeastOneElected for 3 Process, 7 Time
// This should not find any counterexample

pred looplessPath {
  no disj t, t': Time | toSend.t = toSend.t' }

// This produces an instance
run looplessPath for 3 Process, 12 Time

// This does not produce an instance
run looplessPath for 3 Process, 13 Time
```

To fix this problem, we can force progress by insisting that whenever some process has a nonempty identifier pool, some process (not necessarily the same one) must make a move. We write this as a predicate

and then condition the assertion on this predicate holding:

Here's how it works. We write a predicate whose instances are *loopless paths*—traces in which a given state is visited at most once. The behavior of our protocol depends only on the identifier pools, so we'll regard two time instants in a trace as having equivalent states when their pools are equal:

// Therefore, we can conclude that a scope of 12 for Time is
// sufficient to reach all states of the protocol for a three-node ring.