MFES - Métodos Formais em Engenharia de Software

Alloy

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

File system

```
// A File system has objects
abstract sig Object {}

// Objects can be files or directories
sig Dir, File extends Object {}

// A file system has objects
// A file system has one root
// Every object inside a file system has a parent (Dir);
sig FS {
   objects : set Object,
   root : Dir,
   parent : Object -> lone Dir
}
```


File System

```
sig FS {
   objects : set Object,
   root : Dir,
   parent : Object -> lone Dir
}
```

```
// all objects have parents except the root
// the root doesn't have parent
// not cyclic
// the root belogs to the objects of the file system
// the parent relationship belogs to the file system
pred Inv [fs : FS] {
   all x : fs.objects - fs.root | some x.(fs.parent)
   no fs.root.(fs.parent)
   no x : fs.objects | x in x.^(fs.parent)
   fs.root in fs.objects
   fs.parent in fs.objects -> fs.objects
}
run Inv for 3 but exactly 1 FS
```


File system

```
sig FS {
   objects : set Object,
   root : Dir,
   parent : Object -> lone Dir
}
```

```
// pre-conditions:    d is a directory inside the FS
//                    d has no objects inside
//                    d is different from the root
pred rmdir [fs : FS, d : Dir, fs' : FS] {
   d in fs.objects
   no fs.parent.d
   d not in fs.root

   fs'.objects = fs.objects - d
   fs'.parent = fs.parent - (d -> Dir)
   fs'.root = fs.root
}
```

File System

```
pred rmdir_test [fs : FS, d : Dir, fs' : FS] {
   Inv[fs]
   rmdir[fs,d,fs']
}
run rmdir_test for 4 but 2 FS

// If fs is consistent, fs' obtained from fs after removing d
// is also consistent
check {
   all fs, fs' : FS, d : Dir | Inv[fs] && rmdir[fs,d,fs'] =>
   Inv[fs']
} for 3 but 2 FS
```


Properties

- ***Satisfability***: there is a solution
  - run
- ***Preservation of invariants***: the result state of the operations remains correct
  - all s,s': State & Inv(s) and operN(s,s') => Inv(s')
- ***Security***: all reachable states are secure
  - all s:State & Secure(s)
- ***Deadlock***: it may be interesting to show the presence or the absence of deadlocks ("*states without next state*": a situation wherein two or more competing actions are each waiting for the other to finish, and thus neither ever does)

Properties

- ***Livelocks***: ("*states that are never reached*": states of the processes involved in the livelock constantly change with regard to one another, none progressing). Example: when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.
- ***Determinism***: only one possible next state
- ***Inevitability***: states that cannot be avoided

The use of Integers

- Alloy users commonly ask for integers, but usually they do not need the full power of Alloy's built-in integers. There are essentially three levels of integer use:
  - using an ordered set, allowing comparisons but not addition. Use the **models/util/ordering.als** module.
  - using natural (non-negative) numbers, allowing addition but not negatives. Use the **models/util/natural.als**
  - using integers, allowing negatives. Use **models/util/integer.als** contains utility functions and predicates for using Int. Offers simple functions for add, subtract, negate, etc., and common predicates for inequalities.

Ordered set

- Using the ordering module, one might write, for example, the following:

```
module sample
open util/ordering[state] as ord //use the parametric ordering module
sig state {tomorrow: lone state}
one sig today in state {}
fact {today = ord/first} //clarify which ordering you are refering to
fact {all s: state | s.tomorrow = ord/next[s]}
```


Ordered set

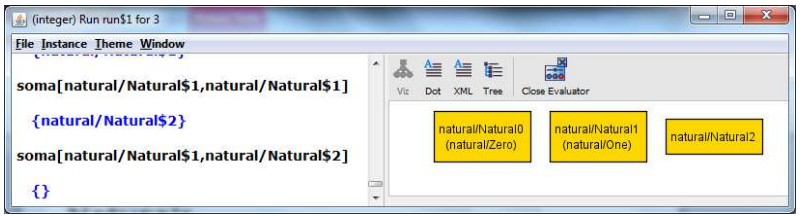
- fun first: one elem { Ord.First }
- fun last: one elem { elem - (next.elem) }
- fun prev : elem->elem { ~(Ord.Next) }
- fun next : elem->elem { Ord.Next } // all pairs where the 2nd is the next of the 1st
- fun prevs [e: elem]: set elem { e.^(~(Ord.Next)) }
- fun nexts [e: elem]: set elem { e.^(Ord.Next) }
- pred lt [e1, e2: elem] { e1 in prevs[e2] }
- pred gt [e1, e2: elem] { e1 in nexts[e2] }
- pred lte [e1, e2: elem] { e1=e2 || lt [e1,e2] }
- pred gte [e1, e2: elem] { e1=e2 || gt [e1,e2] }
- fun larger [e1, e2: elem]: elem { lt[e1,e2] => e2 else e1 }
- fun smaller [e1, e2: elem]: elem { lt[e1,e2] => e1 else e2 }
- fun max [es: set elem]: lone elem { es - es.^(~(Ord.Next)) }
- fun min [es: set elem]: lone elem { es - es.^(Ord.Next) }

Naturals

- Using natural (non-negative) numbers, allowing addition but not negatives.
- natural.als uses the ordering module to represent the natural numbers, including zero. It supports add, subtract, and even multiply and divide.
- If you only need nonnegative numbers, it is much more efficient than using Int.
- One nice side effect of using the ordering module is that the atoms Natural_0, Natural_1, Natural_2, . . . correspond to the integers 0, 1, 2, and so on.

Naturals

```
open util/natural
fun soma[a: Natural, b: Natural]:Natural{
        {x:Natural | x= natural/add[a,b]}
}
run {} for 3
```



Executado com a versão 4.1.10 do Alloy

Naturals

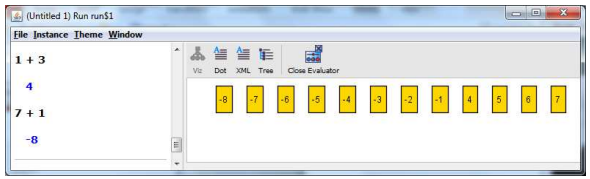
```
fun inc [n: Natural] : lone Natural { ord/next[n] }
fun dec [n: Natural] : lone Natural { ord/prev[n] } // returns n - 1
fun add [n1, n2: Natural] : lone Natural { // returns n1 + n2
  {n: Natural | #ord/prevs[n] = #ord/prevs[n1] + #ord/prevs[n2]}
}
fun sub [n1, n2: Natural] : lone Natural { // returns n1 - n2
  {n: Natural | #ord/prevs[n1] = #ord/prevs[n2] + #ord/prevs[n]}
}
fun mul [n1, n2: Natural] : lone Natural { // returns n1 * n2
  {n: Natural | #ord/prevs[n] = #(ord/prevs[n1]->ord/prevs[n2])}
}
fun div [n1, n2: Natural] : lone Natural { // returns n1 / n2
  {n: Natural | #ord/prevs[n1] = #(ord/prevs[n2]->ord/prevs[n])}
}
```


Naturals

- pred gt [n1, n2: Natural] { ord/gt [n1, n2] } // greater than
- pred lt [n1, n2: Natural] { ord/lt [n1, n2] } // less than
- pred gte [n1, n2: Natural] { ord/gte[n1, n2] } // greater than or equal to
- pred lte [n1, n2: Natural] { ord/lte[n1, n2] } // less than or equal to
- fun max [ns: set Natural] : lone Natural { ord/max[ns] } //returns the maximum integer in ns
- fun min [ns: set Natural] : lone Natural { ord/min[ns] } // returns the minimum integer in ns

Integers

- Using integers, allowing negatives. **integer.als** contains utility functions and predicates for using Int. Offers simple functions for add, subtract, negate, etc., and common predicates for inequalities.
- The main complication is the distinction between "Int" and "int". "Int" is the set of integers that have been instantiated, while "int" returns the value of an Int. You have to say "int i" to be able to add, subtract, and compare "Int"s.



Executado com a versão 4.1.10 do Alloy

Integers example

```
module numbers
//it's all about keeping int and Int straight
fact ThreeExists {
//there is some integer whose value is 3
      some x: Int | int x = 3
}
fun add(a,b:Int):Int {
//give me the set of integers such that their value is the sum of
// the values of a and b. This will be a singleton set, but I
// still used a set comprehension.
      {i: Int | int i = int a + int b}
}

//there should be a solution in this scope
run add for 10 but 3 int expect 1
```

Integers: some warnings

- Some warnings about using integers:
  - they don't scale nearly as well as ordered sets so don't use them if you can get away with ordering or naturals. The ordering module is much faster and easier to use.
  - integers are scoped differently than other signatures; by default, integers have 4 bits and thus range from -8 to +7. To change the number of integers, specify the number of bits representing each integer (i.e., how large their values can get) changing the scope of int. If you do not explicitly change the scope of int, it will remain at the default of 4.
  - For example, you might say the following:

    run show for 3 but 5 int

    which says that most things have a size of at most 3, but there are integers each represented by 5 bits. The number of Int atoms is always equal to 2^(integer bitwidth).

Integers

- fun add [n1, n2: Int] : Int { n1 fun/add n2 }
- fun sub [n1, n2: Int] : Int { n1 fun/sub n2 }
- fun mul [n1, n2: Int] : Int { n1 fun/mul n2 }
- fun div [n1, n2: Int] : Int { n1 fun/div n2 }
- fun rem [n1, n2: Int] : Int { n1 fun/rem n2 }
- fun negate [n: Int] : Int { 0 - n }
- pred eq [n1, n2: Int] { int[n1] = int[n2] }
- pred gt [n1, n2: Int] { n1 > n2 }
- pred lt [n1, n2: Int] { n1 < n2 }
- pred gte [n1, n2: Int] { n1 >= n2 }
- pred lte [n1, n2: Int] { n1 <= n2 }

Integers

- pred zero [n: Int] { n = 0 }
- pred pos [n: Int] { n > 0 }
- pred neg [n: Int] { n < 0 }
- pred nonpos [n: Int] { n <= 0 }
- pred nonneg [n: Int] { n >= 0 }
- fun signum [n: Int] : Int { n<0 => (0-1) else (n>0 => 1 else 0) }

Integers

```
/* returns the ith element (zero-based) from the set s
 * in the ordering of 'next', which is a linear ordering
 * relation like that provided by util/ordering
 */
fun int2elem[i: Int, next: univ->univ, s: set univ] : lone s {
  {e: s | #^next.e = int i }
}

/ * returns the index of the element (zero-based) in the
 * ordering of next, which is a linear ordering relation
 * like that provided by util/ordering
 */
fun elem2int[e: univ, next: univ->univ] : lone Int {
  Int[#^next.e]
}
```

Integers

- fun max:one Int { fun/max }
- fun min:one Int { fun/min }
- fun next:Int->Int { fun/next }
- fun prev:Int->Int { ~next }
- fun max [es: set Int]: lone Int { es - es.^prev }
- fun min [es: set Int]: lone Int { es - es.^next }
- fun prevs [e: Int]: set Int { e.^prev }
- fun nexts [e: Int]: set Int { e.^next }
- fun larger [e1, e2: Int]: Int { let a=int[e1], b=int[e2] | (a<b => b else a) }
- fun smaller [e1, e2: Int]: Int { let a=int[e1], b=int[e2] | (a<b => a else b) }