MFES – Métodos Formais em Engenharia de Software

Alloy

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

Logic operators

not	**!**	negation
and	**&&**	conjunction
or	**||**	disjunction
implies	**=>**	implication
else	**,**	alternative
iff	**<=>**	bi-implication

Quantifiers

all x:e | F	F holds for every x in e;
some x:e | F	F holds for some x in e;
no x:e | F	F holds for no x in e;
lone x:e | F	F holds for at most one x in e;
one x:e | F	F holds for exactly one x in e.

Quantifiers (examples)

```
abstract sig Target {}
sig Name extends Target {}
sig Address extends Target {}
sig Book { address: Name -> Target }
```

some Name
says that the set of names is not empty;

some address
says that the address book is not empty: there is some pair mapping a name to an address;

no (Book.address.Address - Name)
says that nothing is mapped to addresses except for names;

all n: Name | lone n.(Book.address)
says that every name maps to at most one address (more succinctly than in the previous example);

all n: Name | one n.(Book.address) or no n.(Book.address)
says the same thing.

Quantifiers (examples)

```
abstract sig Target {}
sig Name extends Target {}
sig Address extends Target {}
sig Book { address: Name -> Target }
```

- some n:Name, a:Address | a in n.(Book.address)
  - Some name maps to some address – address book not empty
- no n:Name | n in n.^(Book.address)
  - No name can be reached by lookups from itself – address book acyclic
- all n:Name | lone a:Address | a in n.(Book.address)
  - Every name maps to at most one address – address book is functional
- all n:Name | no disj a,a': Address | (a + a') in n.(Book.address)
  - No name maps to two or more distinct addresses – same as above

Comprehensions

```
abstract sig Target {}
sig Name extends Target {}
sig Address extends Target {}
sig Book { address: Name -> Target }
```

- {n:Name | no n.^(Book.address) & Address}
  - Set of names that don't resolve to any actual addresses
- {n:Name, a:Address | n->a in ^(Book.address)
  - Binary relation mapping names to reachable addresses

Let expression and restrictions

```
let x = e | A
```

Examples:

```
all a: Alias |
  let w = a.workAddress |
    a.address = if some w then w else a.homeAddress
```

or

```
all a: Alias |
  a.address =
    let w = a.workAddress |
      if some w then w else a.homeAddress
```


Sintaxe

- ***sig*** – defines a set
- ***fact*** – expresses a truth
- ***pred*** – predicate
- ***fun*** - function
- ***assert*** – property
- ***check*** – check property within certain limits
- ***run*** – search for instance of predicate within scope

Example

(

Modeling “ceilings and floors”

- sig Platform {}
  - There are “Platform” things
- sig Man {ceiling, floor: Patform}
  - Each Man has a ceiling and floor Platform
- fact {all m:Man | some n:Man | Above[n,m]}
  - “One Man’s Ceiling is Another Man’s Floor”
- pred Above(m, n:Man) {m.floor = n.ceiling}
  - Man m is “above” Man n if m’s floor is n’s ceiling

Checking “ceilings and floors”

```
assert BelowToo {
     all m:Man | some n:Man | Above[n,m]
}
```

“One Man’s Floor Is Another Man’s Ceiling?”

Check BelowToo for 2

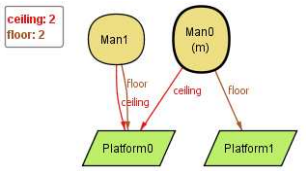
check “One Man’s Floor Is Another Man’s Ceiling”

counterexample with 2 or less platforms and men?

Clicking “Execute” ran this command

- counterexample found, shown in graphic

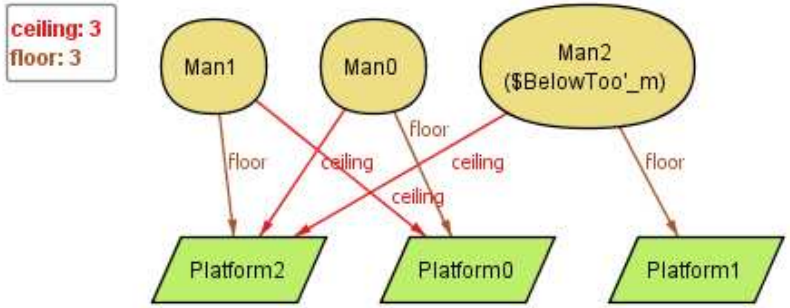
Counterexample to BelowToo



"Ceilings and floors"

- pred Geometry {no m: Man | m.floor = m.ceiling}
- assert BelowToo' { Geometry => (all m: Man | some n: Man | m.Above[n]) }
- check BelowToo' for 2 expect 0
- check BelowToo' for 3 expect 1

Counterexample to BelowToo'



"Ceilings and floors"

```
pred NoSharing {
        no m,n: Man | m!=n &&
        (m.floor = n.floor || m.ceiling = n.ceiling)
}

assert BelowToo" { NoSharing => (all m: Man | some n: Man |
                                  m.Above[n]) }
check BelowToo" for 6 expect 0
check BelowToo" for 10 expect 0
```


Example

)