MFES – Métodos Formais em Engenharia de Software

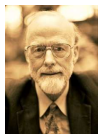
Alloy

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

Some thoughts…

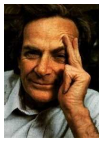
- I conclude there are two ways of constructing a software design: one way is to make it so simple there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

  – Tony Hoare [Turing Award Lecture, 1980]



- The first principle is that you must not fool yourself, and you are the easiest person to fool.

  – Richard P. Feynman



Some thoughts…

- "The core of software development is the design of abstractions"
- "An abstraction is not a module, or an interface, class, or method; it is a structure, pure and simple – an idea reduced to its essential form"
- "I use the term 'model' for a description of a software abstraction"

Daniel Jackson

Some thoughts…

- "Simplicity does not precede complexity, but follows it".

Allan Perlis

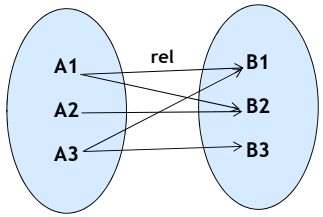
Alloy: four key ideas…

- 1) everything is a relation
- 2) non-specialized logic
- 3) counterexamples & scope
- 4) analysis by SAT (satisfability problem)

The satisfiability problem (SAT) is a decision problem, whose instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: given the expression, is there some assignment of *TRUE* and *FALSE* values to the variables that will make the entire expression true?

1) Everything is a relation

Column A	Column B
A1	B1
A2	B2
A3	B3
A1	B2
A3	B1



{(A1,B1),(A2,B2),(A3,B3),(A1,B2),(A3,B1)}

Alloy:

```
sig B {}
sig A { rel: set B  }
```


2) Non-specialized logic

Three logics into one!

- **Navigation expression style**

  all n: Name, d, d': Address |
  n -> d in address and n -> d' in address
  implies d = d'

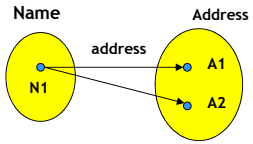
- **Relational calculus style**

  all n: Name | lone n.address

- **Predicate calculus style**

  no ~address.address - iden

Counterexample:



address = {(N1,A1),(N1,A2)}
~address = {(A1,N1),(A2,N1)}
~address.address =
{(A1,A1),(A1,A2),(A2,A1),(A2,A2)}
~address.address - iden =
{(A1,A2),(A2,A1)}

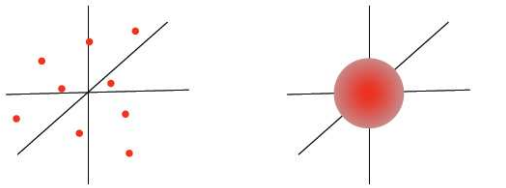
2) Non-specialized logic

"everybody loves a winner"

- Predicate logic
  - ∀w | Winner(w) => ∀p | Loves(p,w)
- Relational calculus
  - Person x winner ⊆ loves
- **Alloy logic** – any way you want
  - all p:Person, w:winner | p -> w in loves
  - Person -> winner in loves
  - all p:Person | winner in p.loves

3) Couterexamples & scope

- Many bugs have small couterexamples
- … and models often have many bugs
- Many more cases than traditional testing



testing:
a few cases of arbitrary size

scope-complete:
all cases within small scope

4) Analysis by SAT

- SAT, the quintessential hard problem (Cook, 1971)
  - > SAT is hard, so reduce SAT to your problem
- SAT, the universal constraint solver (Kautz, Selman et al 1990's)
  - > SAT is easy, so reduce your problem to SAT
  - > solvers: Chaff (Malik), Berkmin (Goldberg & Novikov), others

Stephen Cook

Yakov Novikov

Relations vs Functions

- Funtions / Injectives ?

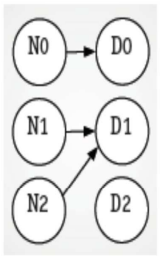


FIG. 3.3 Functional but not injective.

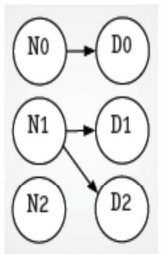


FIG. 3.4 Injective but not functional.

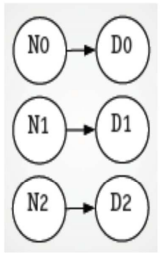


FIG. 3.5 Functional and injective.

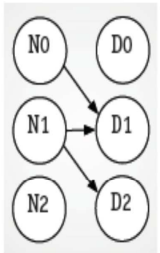


FIG. 3.6 Neither functional nor injective.

The special ones

none	Empty set
univ	Universal set
iden	Identity relation

Distrito = {(Porto), (Lisboa)}

Freguesia = {(Ramalde),(Paranhos)}

none = {}

univ = {(Porto),(Lisboa),(Ramalde),(Paranhos)}

iden = (Porto,Porto),(Lisboa,Lisboa),(Ramalde,Ramalde),(Paranhos,Paranhos)}

Relations

- Sets are unary (1 column) relations

  Name = { (N0), (N1), (N2)} Addr = { (A0), (A1), (A2)} Book = { (B0), (B1)}

- Scalars are singleton sets

  myName = {(N1)}
  yourName = {(N2)}
  myBook = {(B0)}

- Binary relation

  names = {(N0, B0), (N1, B0), (N2, B1)}

- Ternary relation

  addrs = {(N0, B0, A0), (N1, B0, A1), (N1, B1, A2), (N2, B1, A2) }

Relations

- Sets are unary (1 column) relations

  Name = { (N0), (N1), (N2)} Addr = { (A0), (A1), (A2)} Book = { (B0), (B1)}

Alloy:

```
sig N {}
sig A {}
sig B {}
```

- Scalars are singleton sets

  myName = {(N1)}
  yourName = {(N2)}
  myBook = {(B0)}

- Binary relation

  names = {(N0, B0), (N1, B0), (N2, B1)}

Alloy:

```
sig N {
   names: B
}
```

- Ternary relation

  addrs = {(N0, B0, A0), (N1, B0, A1), (N1, B1, A2), (N2, B1, A2) }

Alloy:

```
sig N {
   addrs: B -> A
}
```


Set declarations

x: ***m*** e

x: e <=> x: **one** e

set	any number
one	exactly one
lone	zero or one
some	one or more

RecentlyUsed: set Name

RecentlyUsed is a subset of the set *Name*

senderAddress: Addr (= senderAddress: *one* Addr)

senderAddress is a singleton subset of *Addr*

senderName: lone Name

senderName is either empty or a singleton subset of *Name*

receiverAddresses: some Addr

receiverAddresses is a nonempty subset of *Addr*

Relations

- addrs = {(B0, N0, A0), (B0, N1, A1),(B1, N1, A2), (B1, N2, A2)}

B0	N0	A0
B0	N1	A1
B1	N1	A2
B1	N2	A2

size = 4

arity = 3

- Rows are unordered
- Columns are ordered but unnamed
- All relations are first-order
  - relations cannot contain relations, no sets of sets

Multiplicities

A m -> m B	
set	any number
one	exactly one
some	at least one
lone	at most one

Bestiary

A lone -> B	A -> some B	A -> lone B	A some -> B
Left-unique injective	Left-total	Right-unique Partial function	Right-total Surjective

A lone -> some B	A -> one B	A some -> lone B
Concretization!	function	abstraction

A lone -> one B	A some -> one B
injection	surjection

A one -> one B
bijection

For each kind of relation, draw a counter-example

Cardinalities

`#r`	*number of tuples in r*
`0,1,...`	*integer literal*
`+`	*plus*
`-`	*minus*

`=`	*equals*
`<`	*less than*
`>`	*greater than*
`=<`	*less than or equal to*
`>=`	*greater than or equal to*

```
sum x: e | ie
```
sum of integer expression ie for all singletons x drawn from e

```
all b: Bag | #b.marbles =< 3
```
all bags have 3 or less marbles

```
#Marble = sum b: Bag | #b.marbles
```
the sum of the marbles across all bags equals the total number of marbles

Set operators

Set operators	
in	Subset
+	Union
=	Equality
&	Intersection
-	Difference

Set operators - examples

+	*union*
&	*intersection*
-	*difference*
in	*subset*
=	*equality*

```
Name  = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecentlyUsed = {(N0), (N2)}

Alias + Group = {(N0), (N1), (N2)}
Alias & RecentlyUsed  = {(N2)}
Name - RecentlyUsed   = {(N1)}
RecentlyUsed in Alias = false
RecentlyUsed in Name  = true
Name = Group + Alias  = true
```

```
greg = {(N0)}
rob = {(N1)}

greg + rob   = {(N0), (N1)}
greg = rob   = false
rob in none  = false
```

```
cacheAddr = {(N0, A0), (N1, A1)}
diskAddr = {(N0, A0), (N1, A2)}

cacheAddr + diskAddr = {(N0, A0), (N1, A1), (N1, A2)}
cacheAddr & diskAddr = {(N0, A0)}
cacheAddr = diskAddr = false
```

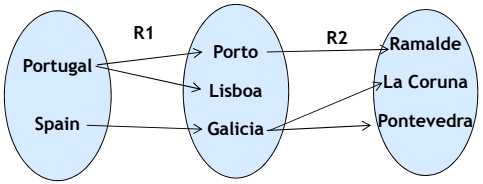

Relational operators

Relational operators	
.	(Dot) join
[]	(Box) join
->	Cartesian product
<:	Domain restriction
:>	Range restriction
++	Override
~	Transpose
^	Transitive closure
*	Reflexive-transitive closure

Join

Key operator is **dot join**

- relational join
- field navigation
- function application



R1 = {(Portugal,Porto),(Portugal,Lisboa),(Spain,Galicia)}

R2 = {(Porto,Ramalde),(Galicia,La Coruna),(Galicia,Pontevedra)}

R1.R2 = {(Portugal,Ramalde),(Spain,La Coruna),(Spain,Pontevedra)}

Join

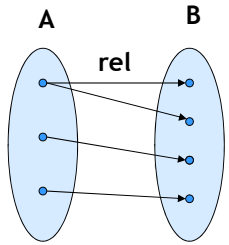
e1 [e2]

has the same meaning as

e2.e1

Binary relation

sig A { rel: some B }



Codomain: **A.rel = rel[A]**

Domain: **rel.B**

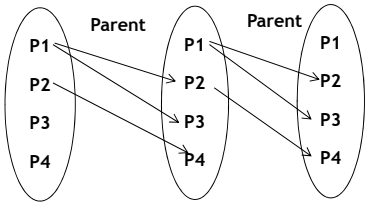
Join

R.S[x] = x.(R.S)

Person = {(P1),(P2),(P3),(P4)}

Parent = {(P1,P2),(P1,P3),(P2,P4)}

Me = {(P1)}



Me.Parent = {(P2),(P3)}

Parent.Parent[me] = {(P4)}

Person.Parent = {(P2),(P3),(P4)}