micromodels of software

declarative modelling and analysis with Alloy

lecture 2: a relational logic

Daniel Jackson
MIT Lab for Computer Science
Marktoberdorf, August 2002

the atlantic divide

American school of formal methods
- emphasis on verification algorithms
- eg, SMV, SPIN, Murphi

European school
- emphasis on modelling
- eg, Z, VDM, B

Alloy brings together
- automatic analysis (like SMV)
- logical notation (like Z)

Pittsburgh, home of SMV

Oxford, home of Z

first order effects

Alloy is first order
- to allow exhaustive search

design implications
- no constructors: composites by projection
- no need to distinguish scalars from singleton sets

novel features
- no scalars or sets: all expressions are relation-valued
- generalized relational join operator
- finite interpretation

atoms

structures are built from

› atoms & relations

atoms are

› indivisible

  can't be broken into smaller parts

› immutable

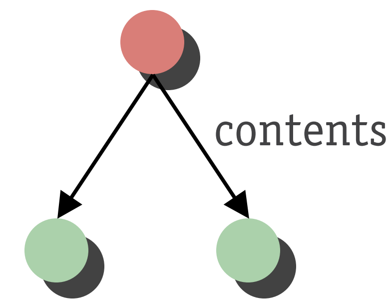
  don't change over time

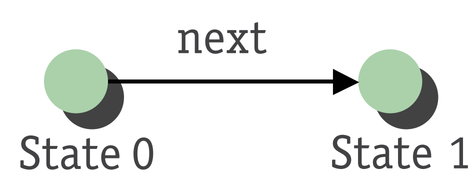
› uninterpreted

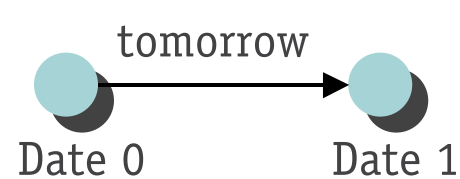
  no built-in properties

what's atomic in the real world?

› very little -- a modelling abstraction







types

universe

› contains all atoms
› a finite (but perhaps big) set
› partitioned into basic types, each a set

```
DATE = {JAN1, JAN2, ..., DEC31}
PERSON = {ALICE, BOB, CAROL}
STATE = {STATE0, STATE1, STATE2}
FILESYSTEM = {FILESYSTEM0, FILESYSTEM2}
```

no subtyping, so

› atoms that share properties share a type

```
Employer = {ALICE}
Employee = {BOB, CAROL}
Employer, Employee in PERSON
```

relations

definition

- a tuple is a list of atoms
- a relation is a set of tuples

```
birthday = {(ALICE,MAY1), (BOB,JAN4), (CAROL,DEC9)}
likes = {(ALICE,BOB), (BOB,CAROL), (CAROL,BOB)}
```

typing

- a relation type is a non-empty list of basic types
- if i-th type is T, then i-th atom in each tuple is in T

```
birthday: (PERSON, DATE)
likes: (PERSON, PERSON)
```

relations as tables

can view relation as table

- atoms as entries, tuples as rows
- order of columns matters, but not order of rows
- can have zero rows, but not zero columns
- no blank entries

example

`birthday = {(ALICE,MAY1), (BOB,JAN4), (CAROL,DEC9)}`

PERSON	DATE
ALICE	MAY1
BOB	JAN4
CAROL	DEC9

dimensions

arity

- number of columns
- relation of arity k is a k-relation
- unary, binary, ternary for k=1, 2, 3
- finite, >0

size

- number of rows
- finite, ≥ 0

    #p   is an integer expression giving the size of p

homogeneity

- relation of type (T, T, …T) is homogeneous
- else heterogeneous

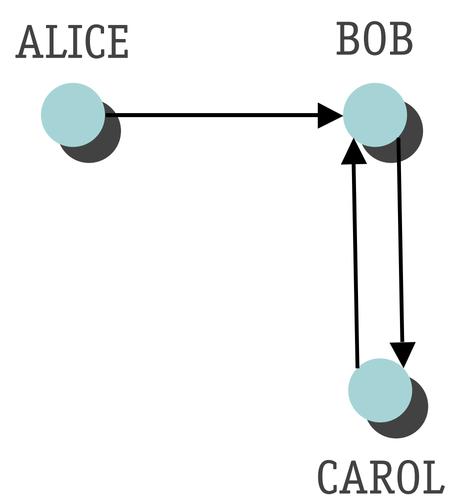
relations as graphs

can view 2-relation as graph

› atoms as nodes
› tuples as arcs

example

`likes = {(ALICE,BOB), (BOB,CAROL), (CAROL,BOB)}`



sets and scalars

sets and scalars

- represented as relations
- set: a unary relation
- scalar: a unary, singleton relation

```
PERSON = {(ALICE), (BOB), (CAROL)}        -- note ()'s!
Employee = {(BOB), (CAROL)}
Employer = {(ALICE)}
Alice = {(ALICE)}
```

unlike standard set theory

- no distinction between

  a, (a), {a}, **{(a)}**

ternary relations

for relationships involving 3 atoms

```
salary: [PERSON, COMPANY, SALARY]
salary = {(ALICE,APPLE,$60k), (BOB,BIOGEN,$70k)}
```

for associating binary relations with atoms

```
birthdayRecords: [BIRTHDAYBOOK, PERSON, DATE]
birthdayRecords =
  {(BB0,ALICE,MAY1), (BB0,BOB,JAN4), (BB1,CAROL,DEC9)}
```

left and right

left and right sets

- left (right) set of p is set of atoms in left-(right-)most column

left and right types

- left (right) type of p is the first (last) basic type of p's type

examples

```
likes = {(ALICE,BOB), (BOB,CAROL), (CAROL,BOB)}
left-set(likes) =  {(ALICE,BOB,CAROL)}
right-set(likes) =  {(BOB,CAROL)}
left-type(likes) = right-type(likes) = PERSON
```

set operators

standard set operators

union	p + q	contains tuples of p and tuples of q
intersection	p & q	contains all tuples in both p and q
difference	p - q	contains tuples in p but not in q

interpretation of +

› for scalars, makes a set	Alice + Bob
› for sets, makes a new set	Employer + Employee
› for relations, combines maps	likes + Alice -> Bob

subset and equality

subset	p in q	q contains every tuple p contains
equality	p = q	p and q contain same set of tuples

product

definition

if p contains (p1,…,pn)

and q contains (q1,…,qm)

then **p -> q** contains (p1,…,pn,q1,…,qm)

puns

for sets s and t, **s->t** is cartesian product

for scalars a and b, **a->b** is tuple

examples

birthday = Alice->May1 + Bob->Jan4 + Carol->Dec9

Employee->Employee in likes

join

definition

if p contains (p1,…,pn-1,pn)

and q contains (q1,…,qm)

and pn = q1

then **p . q** contains (p1,…,pn-1,q2,…,qm)

constraints

arity(p) + arity(q) > 2

right-type(p) = left-type(q)

join, examples

given

```
Alice = {(ALICE)}, bb0 = {(BB0)}
likes = {(ALICE,BOB), (BOB,CAROL), (CAROL,BOB)}
birthday = {(ALICE,MAY1), (BOB,JAN4), (CAROL,DEC9)}
birthdayRecords =
  {(BB0,ALICE,MAY1), (BB0,BOB,JAN4), (BB1,CAROL,DEC9)}
```

we have

```
Alice.likes = {(BOB)}; likes.Alice = {}
likes.birthday = {(ALICE,JAN4), (BOB,DEC9),(CAROL,JAN4)}
bb0.birthdayRecords = {(ALICE,MAY1), (BOB,JAN4)}
Alice.(bb0.birthdayRecords) = {(MAY1)}
```

join, puns

puns

- for set s and binary relation r, **s.r** is image of s under r
- for binary relations p and q, **p.q** is standard join of p and q
- for binary relation r of type (S,T),
  - **S.r** is right-set of r
  - **r.T** is left-set of r

join variants

for non-binary relations, join is not associative

3 syntactic variants of join

p.q = p::q = q[p]

binding power: :: most, then ., then []

p.q::r = p.(q.r)

p.q[r] = r.(p.q)

equivalent expressions

Alice.(bb0.birthdayRecords)

Alice.bb0::birthdayRecords

bb0.birthdayRecords [Alice]

transpose

for relation r: (S,T)

- **~r** contains (b,a) whenever r contains (a,b)
- **~r** has type (T,S)

a theorem

- for set s and binary relation r,
  - **r.s = s.~r**

override

for relations p,q: (S,T)

p++q contains (a,b) whenever

q contains (a,b), or

p contains (a,b) and q does not map a

given

Alice = {(ALICE)}, March3 = {(MAR3)}

birthday = {(ALICE,MAY1), (BOB,JAN4), (CAROL,DEC9)}

we have

birthday ++ Alice->March3 =

{(ALICE,MAR3), (BOB,JAN4), (CAROL,DEC9)}

closure

for relation r: (T,T)

^r = r + r.r + r.r.r + r.r.r.r + …

is smallest **transitive** relation p containing r

***r** = iden[T] + r + r.r + r.r.r + r.r.r.r + …

is smallest **reflexive & transitive** relation p containing r

examples

ancestor = ^parent

reaches = *connects

precedes = ^~next

operator types

if …	then …
p,q: (T1,…,Tn)	p+q, p-q, p&q: (T1,…,Tn), p in q
p: (S1,…,Sn), q: (T1, …,Tm), Sn = T1	p.q: (S1,…,Sn-1,T2,…,Tm)
p: (S,T)	~p: (T,S)
p: (T,T)	*p, ^p: (T,T)
p: (T)	iden[p]: (T,T)
p: (T1,…,Tn)	none[p], univ[p]: (T1,…,Tn)

navigation expressions

from 2-relations and the operators

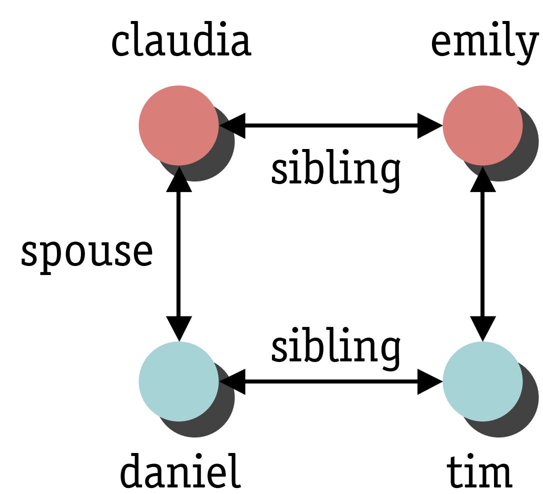
. + ^ * ~

interpret as path-sets

p.q	follow p then q
p+q	follow p or q
^p	follow p once or more
***p**	follow p zero or more times
~p	follow p backwards

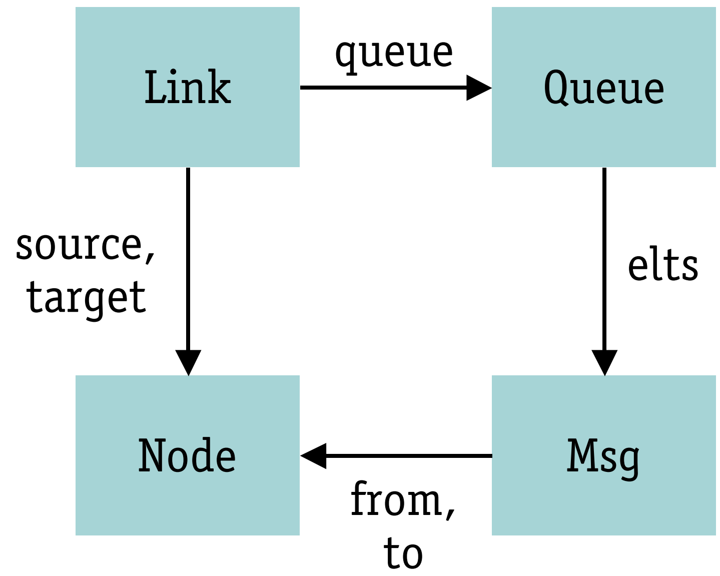
example

cousin = parent.sibling.~parent



daniel.**spouse.sibling** =
daniel.**sibling.spouse**

a navigation example



to say

- all messages queued on links emanating from a node have a 'from' field of that node

we can write

```
all n: Node | n.~source.queue.elts.from = n
```

or equivalently

```
~source.queue.elts.from in iden[Node]
```

logical operators

standard connectives

! F	**not** F	
F **&&** G	F **and** G	**{** F G **}**
F **||** G	F **or** G	
F **=>** G , H	F **implies** G **else** H	
F **<=>** G	F **iff** G	

if-then-else expressions

if F **then** e **else** e′

negated operators

e **!in** e′ , e **!=** e′

set declarations

same meaning as
Employee: ℙ Person
in a other languages,
but first order

form

var : [**set** | **option**] setexpr

meaning

v : e	v **in** e **and** #v = 1
v : **set** e	v **in** e
v : **option** e	v **in** e **and** #v ≤ 1

examples

p : Person	*p is a scalar in Person*
Employee: set Person	*Employee is a subset of Person*
bb : Person -> Date	*not unary, so no scalar constraint*

relation declarations

form

```
var : expr [mult] -> [mult] expr
```

multiplicity symbols

- ? zero or one
- ! exactly one
- \+ one or more

meaning

r: e0 *m* **->** *n* e1

means r **in** e0 -> e1

and *n e1's for each e0,*
m e0's for each e1

examples

r: A ->? B

r is a partial function

r: A ->! B

r is a total function

r: A ?->? B

r is an injective

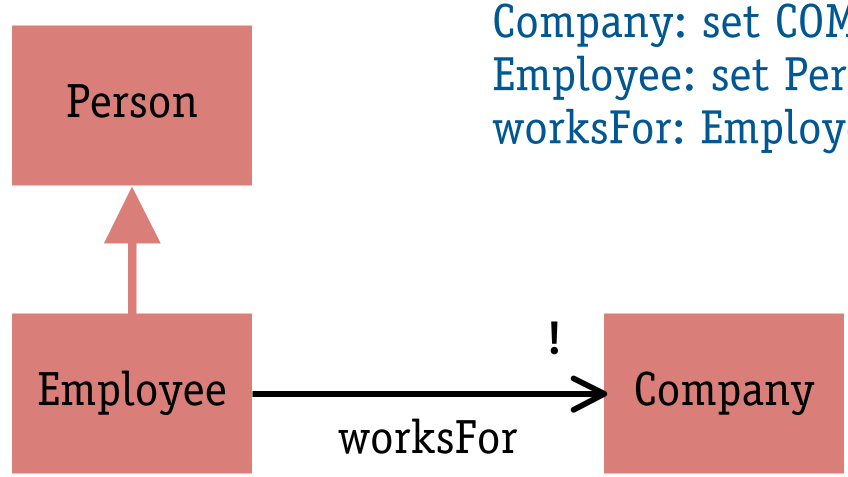
r: A !->! B

r is a bijection

object models

what is an object model?

- set of declarations drawn as graph
- boxes denote sets, arcs relations
- parentless box has implicit type




```
Person: set PERSON
Company: set COMPANY
Employee: set Person
worksFor: Employee ->! Company
```

comprehensions

general form

{ var : setexpr ,… | formula }

meaning

{ v0: e0, v1: e1, … | F }

is the relation containing tuples (a0,a1, …)

such that F holds when v0 = {(a0)}, v1 = {(a1)}, etc

and {(a0)} in e0, {(a1)} in e1, etc

example

sibling = {a, b: Person | a.parents = b.parents && a != b}

quantification

universal quantification

all var : setexpr ,... | formula

meaning

all v0: e0, v1: e1, ... | F

holds iff F holds whenever v0 = {(a0)}, v1 = {(a1)}, etc

and {(a0)} in e0, {(a1)} in e1, etc

example

all a: Person | a **!in** a.parents

other quantifiers

quantifiers

all x: e | F — F holds for **all** 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

sole x: e | F — F holds for **at most one** x in e

one x: e | F — F holds for **exactly one** x in e

note

all v0: e0, v1: e1,… | F *is equivalent to*

  all v0: e0 | **all** v1: e1 | … | F

one v0: e0, v1: e1,… | F *is not equivalent to*

  one v0: e0 | **one** v1: e1 | … | F

quantified expressions

for quantifier Q and expression e, make formula

`Q e`

meaning

some e	e is non-empty	#e > 0
no e	e is empty	#e = 0
sole e	e has at most one tuple	#e ≤ 1
one e	e has one tuple	#e = 1

example

no Man **&** Woman   *no person is both a man and a woman*

sample quantifications

biological constraints

```
all p: Person | one p.mother
no p: Person | p in p.parents
```

cultural constraints

```
all p: Person | sole p.spouse
no p: Person | some p.spouse & p.siblings
```

biblical constraints

```
one eve: Person | Person in eve.*~mother
```

summary: doing more with less

everything's a relation

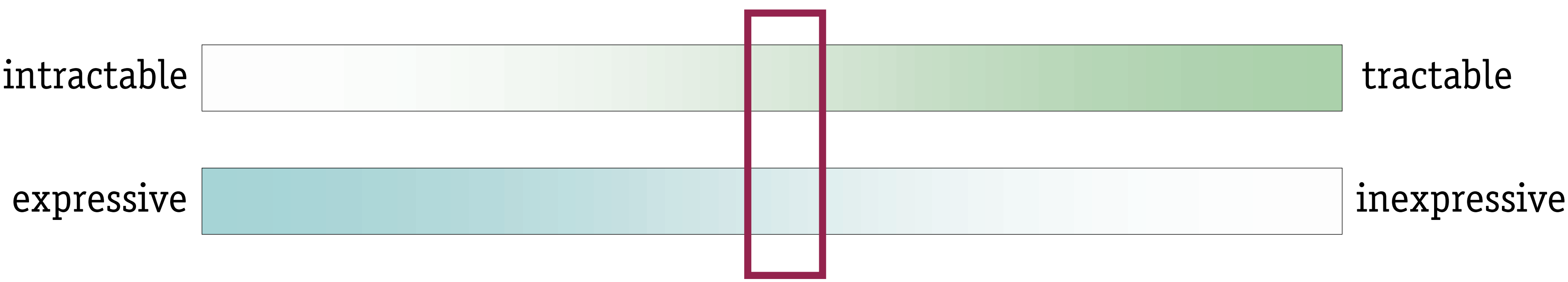
`a->b in r`   *for* (a, b) □ r *and* a □ b □ r

first-order operators

`r : A -> B` means r □ A □ B replaces r □ ℙ(A □ B)

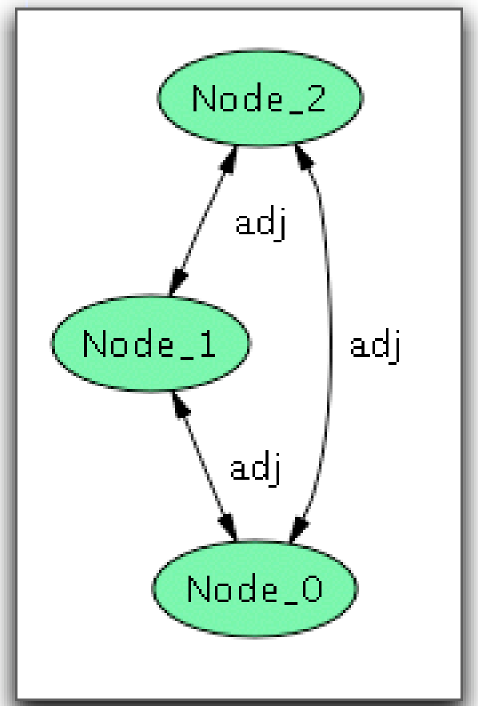
dot operator

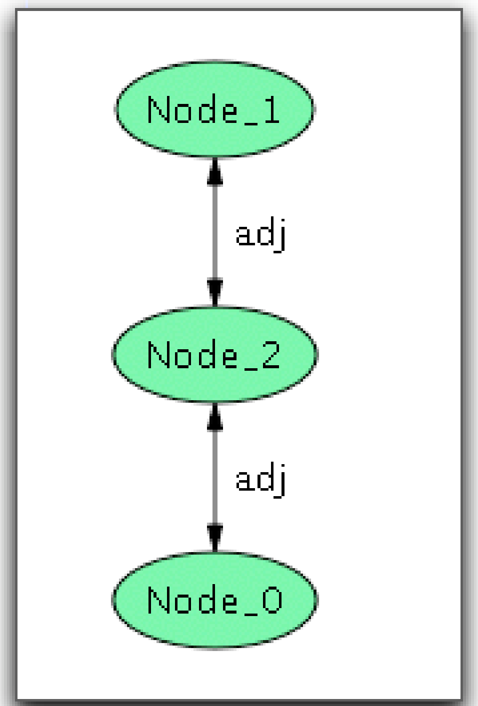
› plays many roles



a challenge

write a constraint
- › on an undirected graph
- › that says it is acyclic





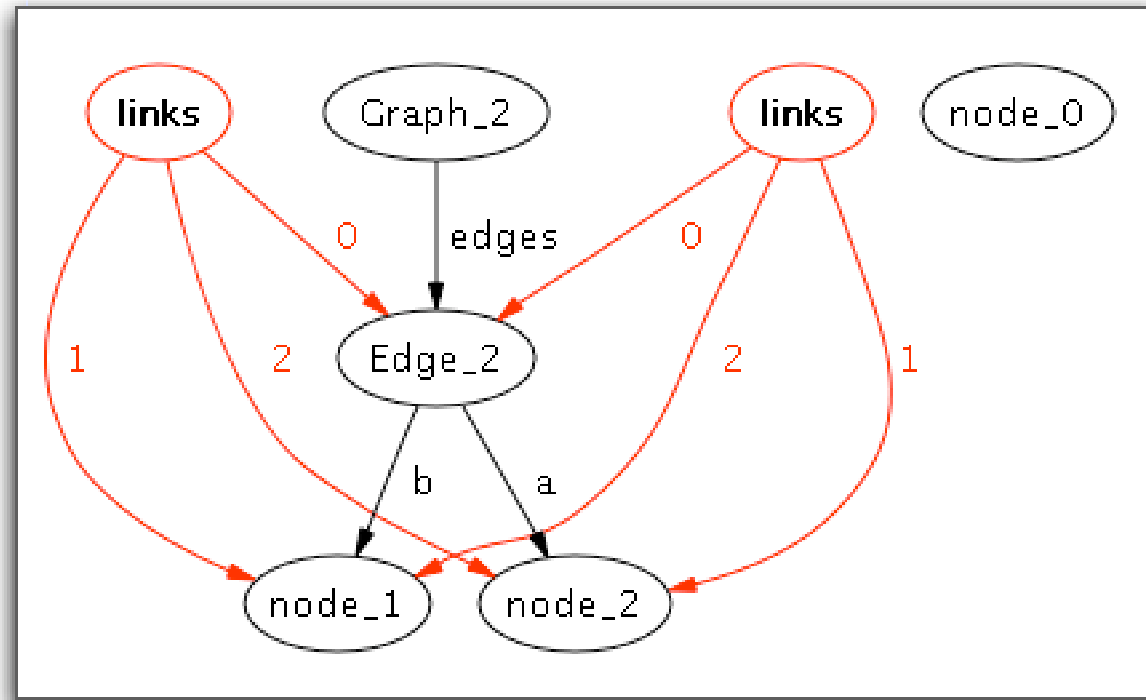
a solution

```
sig Edge[t] {links: t->t}
  {some a,b: t | links = a->b + b->a}

sig Graph[t] {edges: set Edge[t]}

fun Acyclic [t] (g: Graph[t]) {
  no e: Edge[t] |
    let adj = g.edges.links, adj' = (g.edges-e).links |
      *adj = *adj'
  }
```

sample graph



higher-order quantifiers

general form

quantifier decl ,… | formula

modified set expressions

all s: **set** S | F	F holds for all s = S′ where S′ in S
all o: **option** S | F	F holds for all o = S′ where **sole** S′, S′ in S

relational expressions

all x: R | F	F holds for all x = R′ where R′ in R

examples

all p, q, r : T -> T | p.(q.r) = (p.q).r

all p : S -> T, s : **set** S | s.p = ~p.s

model checking

only low-level datatypes
- must encode in records, arrays
- no transitive closure, etc

built-in communications
- not suited for abstract schemes
- fixed topology of processes

modularity
- missing at operation level

culture of model checking
- emphasizes finding showstopper flaws
- but in software, essence is incremental modelling
- keep counters, discard model or vice versa?