A MICRO-MODULARITY MECHANISM

Daniel Jackson, Ilya Shlyakhter & Manu Sridharan
MIT Lab for Computer Science

FSE/ ESEC · Vienna · September 12, 2001

lightweight formalism

software blueprints

- simple & succinct notation
- mouse-click semantic analysis

declarative

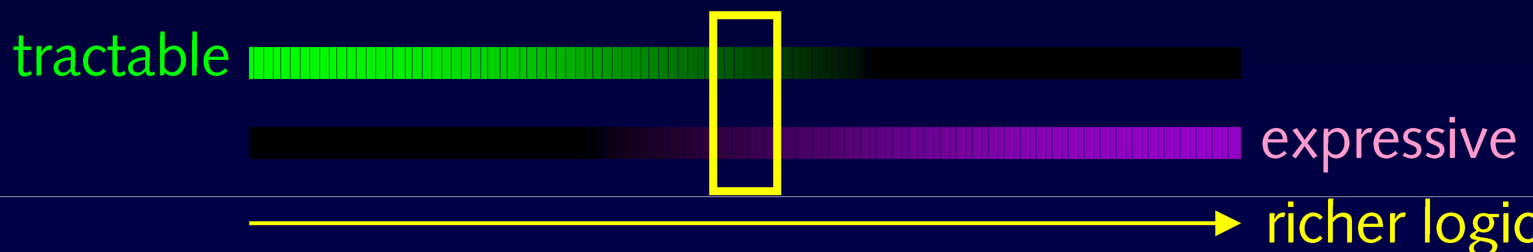
- incremental -- say less, more happens
- implicit -- if I achieved this, then what?

structural

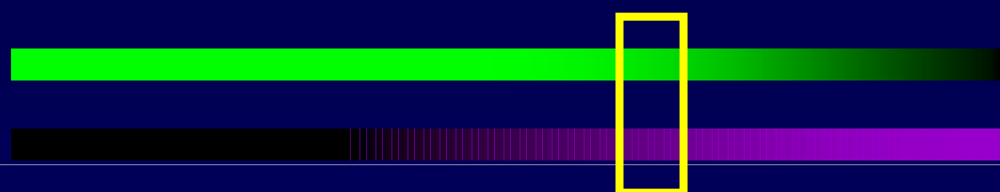
- state itself has graph structure
- high-level expression

progress

inherent conflict



new SAT-based analysis (FSE 2000)



new language (today's talk)

alloy 2k's flaws

kernel limitations
- binary relations
- (no numbers)

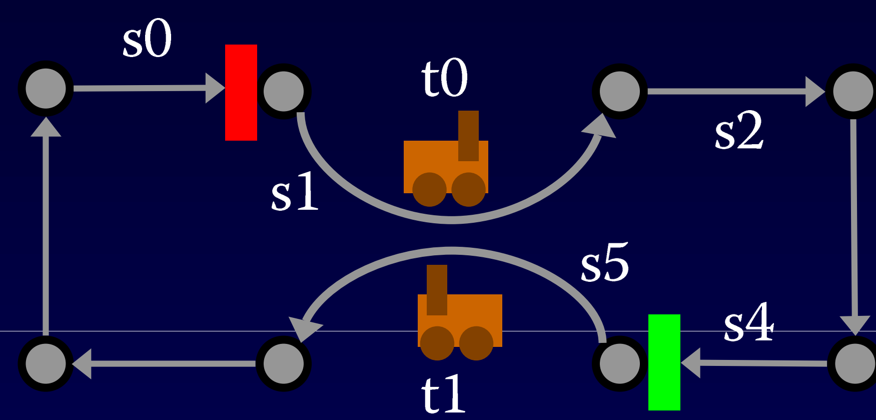
poor structuring mechanisms
- no operation sequencing
- no library of datatypes
- no incremental spec

alloy XP's features

solves these problems
- and even simpler!

key ideas
- one structuring mechanism
  signature
- one type constructor
  relation
- one operator
  join

railway example



segments

Seg = {s0, s1, …}
next = {(s0, s1), (s4, s5),…}
overlaps = {(s1, s5), (s5, s1),…}

trains

Train = {t0, t1, …}
on = {(t0, s1), (t1, s5)}

gates

closed = {s0}

safety

{t0} {s1} {t1} {s5} {s1,s5}

all disj ta, tb: Train | ta.on !in tb.on.overlaps

railway: segments & trains

signature, the modularity mechanism

facts and functions: just formulas

```
module railway

sig Seg {next , overlaps: set Seg}
fact {all s: Seg | s in s.overlaps}
fact {all s1, s2: Seg | s1 in s2.overlaps => s2 in s1.overlaps}

sig Train {}
sig TrainState {on: Train ->! Seg, occupied: set Seg}
fact {all x: TrainState |
    x.occupied = {s: Seg | some t: Train | t.(x.on) = s}
    }

fun Safe (x: TrainState) {
    all disj ta, tb: Train | ta.(x.on) !in tb.(x.on).overlaps
    }
```

dot is navigation op

railway: gates

```
sig GateState {closed: set Seg}

fun GatePolicy (g: GateState, x: TrainState) {
   x.occupied.~next in g.closed
   all s1, s2: Seg |
      some (s1.next & s2.next.overlaps) => sole (s1+s2 - g.closed)
   }
```

loose spec of mechanism

- no need to say when gates close
- just say ‘at most one not closed’

railway: train motion

```
fun TrainsMove (x, x': TrainState, movers: set Train) {
    all t: movers | t.(x'.on) in t.(x.on).next
    all t: Train - movers | t.(x'.on) = t.(x.on)
    }

fun MayMove (g: GateState, x: TrainState, movers: set Train) {
    no movers.(x.on) & g.closed
    }
```

few assumptions about trains

- no need to say how trains move
- just say ‘movers go to next segments’

railway: theorem

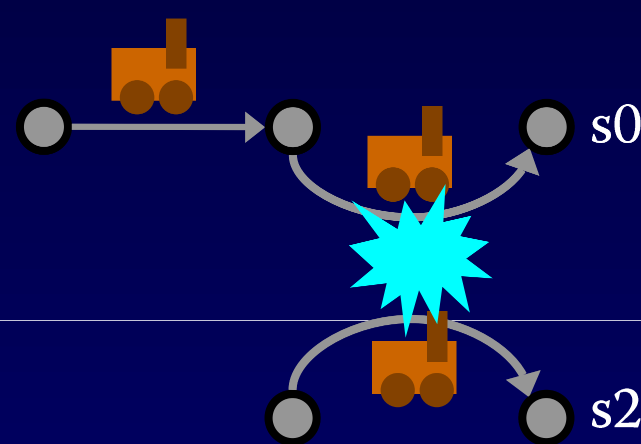
```
assert PolicyWorks {
  all x, x': TrainState, g: GateState, movers: set Train |
    MayMove (g, x, movers) &&
    TrainsMove (x, x', movers) &&
    Safe (x) &&
    GatePolicy (g, x)
      => Safe (x')
  }
```

simple logic

- no special notions of sequential behaviour

bug!

gate policy not strict enough
- train enters segment
  that overlaps with sitting train




```
fun GatePolicy (g: GateState, x: TrainState) {
   x.occupied.overlaps.~next in g.closed
   …}
```

how to reduce to first order logic

interpret structures as atoms

- indivisible & uninterpreted
- signature S introduces basic type S

interpret fields as relations

- global, constant
- sig S {f : T} gives relation f of type $\langle S, T \rangle$

interpret expressions as relations

- set of atoms from basic type S has type $\langle S \rangle$
- scalar is represented as singleton set

interpret dot as relational image

$$[\![p \,.\, q]\!] = \{ (q_2, \ldots q_m) \mid (q_1) \in [\![p]\!] \wedge (q_1, \ldots q_m) \in [\![q]\!] \}$$

signatures

sig Train {}
Train: ⟨Train⟩

sig Seg {next: set Seg}
Seg: ⟨Seg⟩
next: ⟨Seg, Seg⟩

sig TrainState {on: Train -> Seg}
TrainState: ⟨TrainState⟩
on: ⟨TrainState, Train, Seg⟩

typing t.(x.on).next
t : ⟨Train⟩, x : ⟨TrainState⟩
x.on : ⟨Train, Seg⟩
t.(x.on) : ⟨Seg⟩
t.(x.on).next : ⟨Seg⟩

evaluating t.(x.on).next

t = {t0} , x = {x0}
on = { (x0,t0,s0), (x0,t1,s1),
(x1,t0,s1), (x1,t1,s1) }
next = {(s0,s1)}

x.on = {(t0,s0), (t1,s1)}
t.(x.on)= {s0}
t.(x.on).next = {s1}

formulas

kinds of formula

- facts
  - implicitly conjoined to all formulas
- functions
  - explicitly instantiated
  - to run, find a model
- assertions
  - to check, find a counterexample

```
assert PolicyWorks {… MayMove (g, x, ts)… }
fun MayMove (g: GateState, x: TrainState, ts: set Train) {…}
fact {all s: Seg | s in s.overlaps}
```

in the paper …

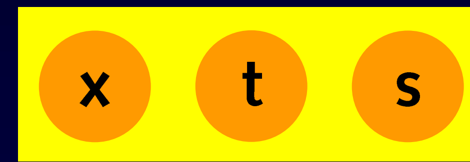
extension mechanism
- for classification of objects
- for adding fields & constraints
- look Ma, no subtypes!

polymorphic data types
- eg, Tree[t], Seq[t]

non-boolean functions
- define implicitly, but invoke as if explicit

flexibility
- illustrations of other idioms
- transitions & traces as objects

related work



objectification

- like object-oriented language
- M.Jackson & Zave's phenomenology
- relational fluents in situation calculus

languages designed for analysis

- model-checking languages
  - complex data structures must be encoded
  - separate languages for artifact & property
- executable specification languages
  - declarative features ruled out
  - user provides test cases

related work: Z

our starting point

- much simpler than OCL, eg
- not first-order, but many Z specs are?

schemas are tricky

- meaning is set of bindings
- binding is finite function
- theta: convert syntax to semantics

schema operators

- involve hiding & renaming
  depend on use of conventions
- see example in paper

experience with alloy

old

- Intentional Naming (Khurshid, ASE 2000)
- COM (Jackson & Sullivan, FSE 2000)
- Role-based Access Control (Zao & Wee, BBN)

new

- Chord peer-peer protocol (Wee)
- network topologies (Shlyakhter, Zakkiudin)
- implicit invocation (Jackson)

Alloy used in courses at

- CMU, Waterloo, Wisconsin, Rochester, Kansas State, Irvine, Georgia Tech, Queen's, Michigan State, Imperial

what else?

http://sdg.lcs.mit.edu/alloy

- beta release available now
- November 2001
  stable release & reference manual

on the horizon

- browser-based visualization
- Chaff, new SAT solver from Princeton
- parallel solver framework
- application to code