Alloy Revisited

Daniel Jackson

IFIP 2.9, Hawks Cay, Florida

February 2001

topics

Flims railway revisited

- in new version of Alloy
- demo with old version

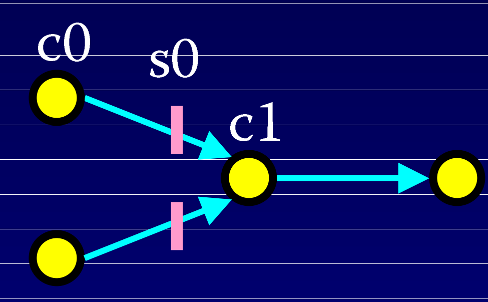
motivations for Alloy

- old ones
- new ones

basic notions

track topology

- segments join connectors

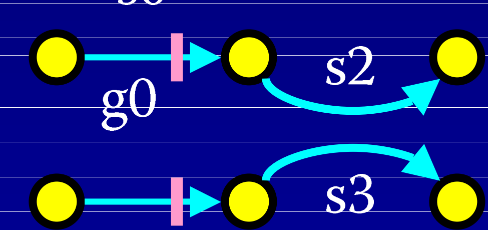


interference

- avoid trains on overlapping segments
- define conflicting segments

gates

- at ends of segments
- train only passes if open



signatures

signature

- denotes a set of individuals
- fields are relations, organized by first type

```
sig Connector {}
sig Segment {
    from, to: Connector,
    nexts, overlaps, conflicts: set Segment,
    gate: option Gate }
sig Gate {}
sig Train {}

sig TrainState {on: Train ->! Segment}
sig GateState {open: set Gate}
```

facts

set operators

relation operators

fact

- an axiom, or global property

```
fact SegmentDefs {
  all s: Segment {
    s.nexts = {t: Segment | s.to = t.from}
    s.conflicts = {t: Segment | some (s.nexts -> t.nexts) & overlaps} - s
    }

fact MinimalOverlaps {
    all s: Segment | s in s.overlaps
    all s, t: Segment | s in t.overlaps => t in s.overlaps
    }
```

image (s.nexts); x-product (->); intersect (&); difference (-); subset (in)

functions

function

- a parameterized formula that can be invoked

```
fun GatePolicy (gs: GateState) {
  all s: Segment | sole (s.conflicts + s).gate & gs.open
  }

fun SafeState (ts: TrainState) {
  all s: Segment | sole s.overlaps.~(ts.on)
  }
```

'quantified expr'

sole e means e has at most one element

more functions

```
fun TrainMotion (pre, post: TrainState, movers: set Train) {
    all t: Train - movers | t.(post.on) = t.(pre.on)
    all t: movers | t.(post.on) in t.(pre.on).nexts
    }

fun MayMove (ts: TrainState, gs: GateState, movers: set Train) {
    movers.(ts.on).gate in gs.open
    }
```

assertions

assertion

- adds intentional redundancy
- analyzer tries to satisfy negation

```
assert SystemSafe {
   all pre, post: TrainState, gs: GateState, movers: set Train |
      SafeState (pre) &&
      GatePolicy (gs) &&
      TrainMotion (pre, post, movers) &&
      MayMove (pre, gs, movers)
      => SafeState (post)
   }
```

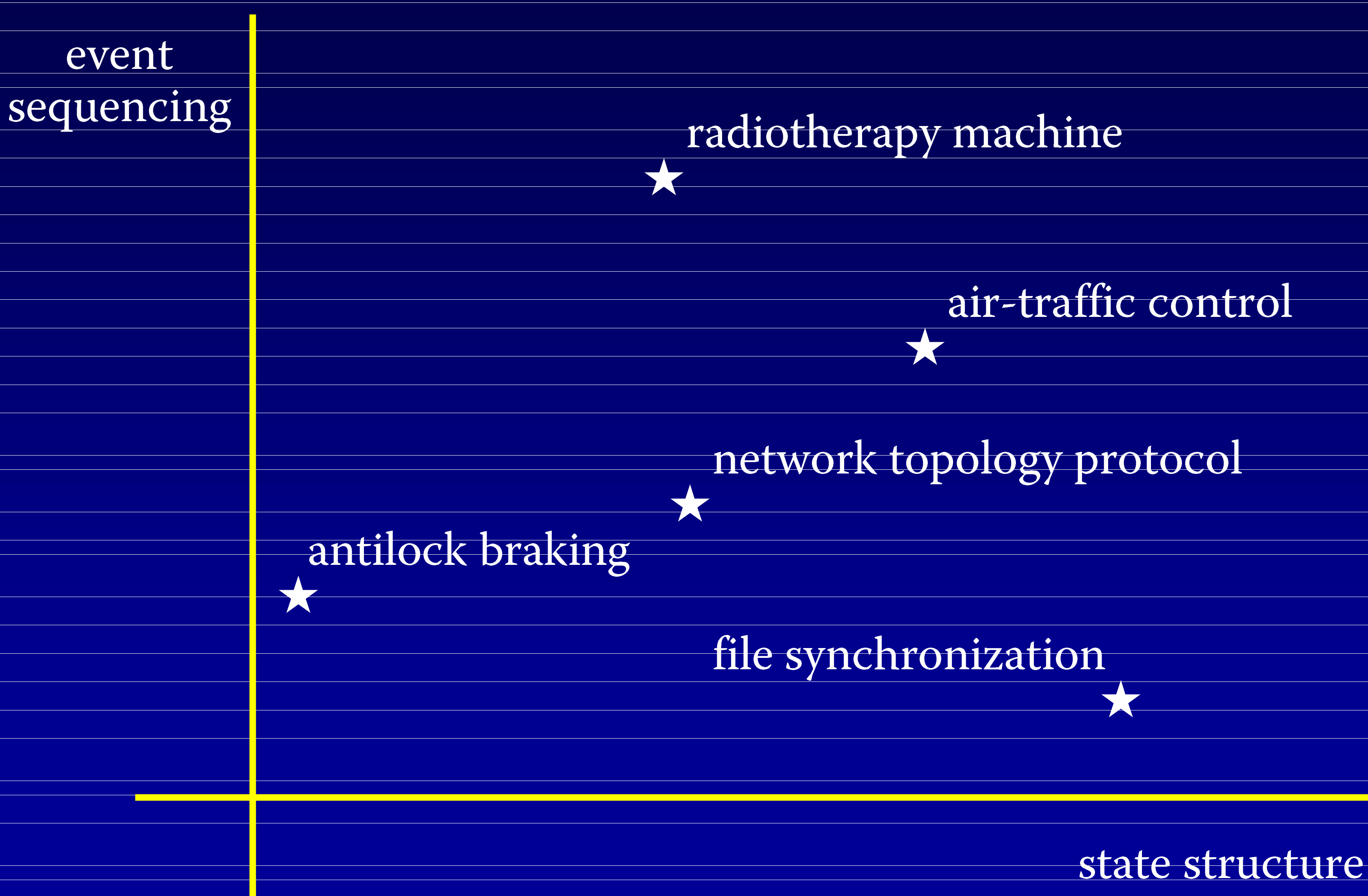
language design motivations

- express structure
  - not 'static vs. dynamic'
- support declarative style
  - property-based description
- be tractable
  - executable $\not\Rightarrow$ operational
- be lightweight
  - in syntax & semantics

expressing structure

- inherent structure
  - eg, track topology
- invented structure
  - 'There is no problem in computer science that cannot be solved by an extra level of indirection (but that usually causes new problems)' *-- David Wheeler*
  - eg, style sheets, domain names, implicit invocation
- not static
  - topologies, styles, names change
  - perhaps 'low frequency'

two axes of complexity



supporting declarative style

declarative spec

- built from properties expressed in a logic
  - eg, safety mechanism, safety requirement, train motion

why be declarative? obvious reasons

- specifying product family
  - eg, what properties must query-interface have?
- factor out safety properties
  - eg, high-power-beam => diffuser-present
- incremental development
  - eg, separate error cases, views, etc

why be declarative?

less obvious reasons

- separate accident from essence
  - eg, what should restore-from-trash do?
- partial specification: replace ops by invariant
  - eg, well-formed resource database vs. spec of registration
- make minimal assumptions about environment
  - eg, allow extra gates, arbitrary train movements
- check minimal mechanism
  - eg, check simple gate rules, not detailed protocol

be tractable

non-compromises

- limit use of declarative constructs
- mention bounds in specification
- make user provide test cases

compromises

- first-order description
  - no spec by minimization
- refutation & simulation, but no proof
- all analysis within bounds
  - #segments, #trains, #msgs, #integers, …

concrete analysis of abstract description

be lightweight

syntax

- no special symbols
- no dependence on graphics
- tiny grammar

semantics

- very few operators
- standard interpretation
- no ‘undefined’

new motivations

specification structure

- incrementality, factoring, hierarchy, sequencing, libraries
- like Z schemas

  but simpler, more semantic, first-order

expressiveness

- arithmetic

  cardinality of sets, +/-/<
- higher-order quantifiers

  refuse analysis when can't skolemize
- general relations

  operators over arbitrary-arity relations

  signatures force this!

uses of signatures (1)

classification of domain objects

- sig Interface {iid: IID, …}
- sig LegalInterface extends Interface {}

library of datatypes

- sig Graph [T] {adj: T -> T}
- sig DAG [T] extends Graph {}
- sig Tree [T] extends DAG {root: T, …}

object-oriented program

- sig AircraftDB {flightRec: Flight -> FlightRec}
- sig FlightRec {plan: Plan, track: Track}

uses of signatures (2)

trace-based analysis

- sig Tick {}
- sig Trace {next: Tick -> Tick, …}
- sig RegistryTrace extends Trace {state: Tick -> DB}
  {all t: Tick | Register (t.state, t.next.state)}
- assert {all x: RegistryTrace | all t: Tick | t.(x.state) …}

globals

- sig Color {}
- static sig Colors {part Red, Yellow, Green: Color}
- … Colors.Red …

summary

signatures

- simpler than schemas
- more semantic in flavour
- still first-order

next

- new version of tool under construction
- case studies
  - network topology propagation (with DERA)
  - security domains (with BBN)
  - air-traffic control (with NASA)

unused bits & pieces

wheeler example: style sheets

- to help document editing
  - tag: Paragraph -> Style
  - val: Style -> Feature -> Value
- to help style editing
  - parent: Style -> Style
- how does user set overrides?
  - f.(s.val) = f.(s.parent.val)?

a weak argument

- complexity theory: easier to check a result than find it
- specification theory: easier to describe a check than an algorithm

notes

better to specify which gates are closed

- then no gate placement policy needed