CSE503: Software Engineering

David Notkin
University of Washington
Department of Computer Science & Engineering
Winter 2002

1

Analysis of model-based specifications

- Given a model-based (Z-like) specification, can we determine if it is inconsistent?
- In particular, can we do for Z-like specifications what we did for model checking: determine if something is not true that we expect to be true

2

Why different?

- Z-like specifications are not suitable for direct model checking
- The primary problem is that the data structures are generally unbounded, taking the problem out of the realm of model checking
- Even simple bounded data structures generally cause massive state space explosions
- Abstraction into a model-checkable problem is feasible, but not generally possible to automate

3

An alternative: counterexample checking

- D. Jackson and C. Damon (Nitpick) suggested an alternative: check a state space of a Z-like specification up to a selected finite bound
- That is, determine if there is an inconsistency within a certain bounded state space
- If a counterexample is reported, one has determined a real error
- If not, one can not distinguish between a consistent specification and one in which the inconsistencies are beyond the chosen bound

4

Why OK?

- This technique is unsound: it may not report counterexamples when they exist
- However
  - The approach is very clear about reporting only counterexamples in the selected bound
  - If it does find counterexamples, they help identify problem early
  - The search space, while bounded, is still large
  - There is an unproven hypothesis that most, or at least many, problems arise in small state spaces (“small scope hypothesis”)

5

Nitpick -> Alcoa

- Nitpick 1996
  - Sets and binary relations, Z-like schema calculus, sequential composition
- Alcoa 2000
  - First-order quantifiers, hierarchical structures, numbers, etc.
  - Performance improvement of at least a factor of two in both the number of relations and the size of the finite bound

6

An example from Jackson

- Rough example of the BART system
  - Investigate topology of railway
  - Investigate placement of gates

7

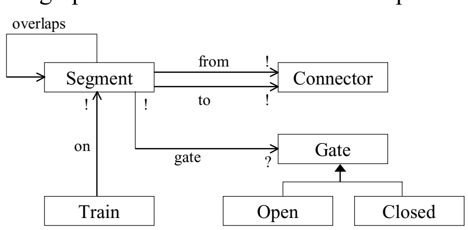
Basic notions

- Segments: capability to use track in one direction
  - A connector at end end
    - `seg1.from = con1`
    - `seg3.from - seg4.from`
- Overlap: model crossings of segments
  - `seg5 in seg6.overlaps` means that `seg5` and `seg6` cross
- Gates: some segments have gates at the end, which may be open or closed
- Train: occupy segments (ignore position)

8

Object model

- A graphical version of a Z-like description



9

Alloy model: declarations

```
model Bart {
  domain {Segment, Connector, Gate, Train}
  state Segments {
    from, to: Segment -> Connector!
    overlaps: Segment -> Segment
    gate: Segment! -> Gate?
    partition Open, Closed: Gate
    on: Train -> Segment!
    succ: Segment -> Segment
    conflicts: Segment -> Segment
}
```

10

An indicative invariant

```
inv Overlaps {
  all s,t | s.from = t.to && s.to = t.from
      -> s in t.overlaps
  all s | s in s.overlaps
}
```

11

A safety condition

- Every segment has at most one train on it and its overlapping segments
  - Could check by theorem proving…or by counterexample checking

```
cond Safety {
  all s | sole(s + s.overlaps).~on
}
```

12

Two definitions

- Semantics of to and from relations

```
def succ {
   all s | s.succ = {t|t.from=s.to}
}
```

- A segment conflicts with another segment if their successors overlap

```
def conflicts {
  all s | s.conflicts =
    {t | some(s.succ & t.succ.overlaps)} - s
}
```

13

Policy invariants

- Place a gate wherever there is a conflict

```
inv GatePlacement {
  all s | some s.conflicts -> some s.gate
}
```

- At most one open gate in a conflicting group

```
inv Policy {
  all s | sole (s.conflicts + s).gate & Open
}
```

14

An operation

- In any step, any number of trains can move; no train goes through a closed gate

```
op TrainsMove(ts: Train) {
  all t: ts | t.on' in t.on.succ
  no (ts.on.gate & Closed)
  all t: Train - ts | t.on = t.on'
}
```

15

Analysis strategy

- Check consistency
  - Ask for instances of states and transitions
- Check consequences
  - Assert implications of invariants
  - Assert properties of invariants (for instance, preservation of invariants)

16

Bug example, implication

- Assert that `conflicts` is symmetric

```
assert ConflictsSym {
  all s,t | s in t.conflicts
    -> t in s.conflicts
}
```

- Alcoa reports a counterexample (with two connectors and two segments)
- Fix by adding constraint on overlaps

```
all s,t | s in t.overlaps -> t in s.overlaps
```

17

Bug example, preservation of invariants

- Assert that the safety condition is preserved

```
assert PolicyWorks {
   all t | TrainsMove(t) && Safety-> Safety'
}
```

- Counterexample returned: a new train was created during the operation…crunch!
- Fix by adding to operation

```
Trains = Trains'
```

18

Underlying technology

- Started using explicit model checking
- Tried symbolic model checking
  – Better in some cases, but highly unpredictable
- Now, SAT solvers

19

Unsound, but useful

- And useful is a very nice property

20