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

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

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”)

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

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
An example from Jackson

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

Basic notions

- Segments: capability to use track in one direction
  - A connector at end end
    - \( \text{seg}.\text{from} = \text{con} \)
    - \( \text{seg}.\text{from} = \text{seg}.\text{from} \)
- Overlap: model crossings of segments
  - \( \text{seg}.\text{in} \text{seg}.\text{overlaps} \) means that \( \text{seg}.\text{and} \text{seg}.\text{cross} \)
- Gates: some segments have gates at the end, which may be open or closed
- Train: occupy segments (ignore position)

Object model

- A graphical version of a Z-like description

Alloy model: declarations

```alloy
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
  }
}
```

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
}
```

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
}
```
Two definitions

- Semantics of to and from relations
  
  \[
  \text{def } \text{succ} \{ \\
  \text{all } s | s.\text{succ} = \{ t | t.\text{from} = s.\text{to} \} \\
  \}
  \]

- A segment conflicts with another segment if their successors overlap
  
  \[
  \text{def } \text{conflicts} \{ \\
  \text{all } s | s.\text{conflicts} = \\
  \{ t | \text{some}(s.\text{succ} \& t.\text{succ}.\text{overlaps}) \} \rightarrow s \\
  \}
  \]

Policy invariants

- Place a gate wherever there is a conflict
  
  \[
  \text{inv } \text{GatePlacement} \{ \\
  \text{all } s | \text{some } s.\text{conflicts} \rightarrow \text{some } s.\text{gate} \\
  \}
  \]

- At most one open gate in a conflicting group
  
  \[
  \text{inv } \text{Policy} \{ \\
  \text{all } s | \text{sole } s.\text{conflicts} \& s.\text{gate} \& \text{Open} \\
  \}
  \]

An operation

- In any step, any number of trains can move; no train goes through a closed gate
  
  \[
  \text{op } \text{TrainsMove}(ts: \text{Train}) \{ \\
  \text{all } t: ts | t.\text{on} \rightarrow t.\text{on}.\text{succ} \\
  \text{no } (ts.\text{on}.\text{gate} \& \text{Closed}) \\
  \text{all } t: \text{Train} - ts | t.\text{on} = t.\text{on}' \\
  \}
  \]

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)

Bug example, implication

- Assert that \text{conflicts} is symmetric
  
  \[
  \text{assert } \text{ConflictsSym} \{ \\
  \text{all } s, t | s \text{ in } t.\text{conflicts} \\
  \rightarrow t \text{ in } s.\text{conflicts} \\
  \}
  \]

- Alcoa reports a counterexample (with two connectors and two segments)

- Fix by adding constraint on overlaps
  
  \[
  \text{all } s, t | s \text{ in } t.\text{overlaps} \rightarrow t \text{ in } s.\text{overlaps} \\
  \]

Bug example, preservation of invariants

- Assert that the safety condition is preserved
  
  \[
  \text{assert } \text{PolicyWorks} \{ \\
  \text{all } t | \text{TrainsMove}(t) \& \text{Safety} \rightarrow \text{Safety}' \\
  \}
  \]

- Counterexample returned: a new train was created during the operation…crunch!

- Fix by adding to operation
  
  \[
  \text{Trains = Trains}' \\
  \]
Underlying technology

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

Unsound, but useful

• And useful is a very nice property