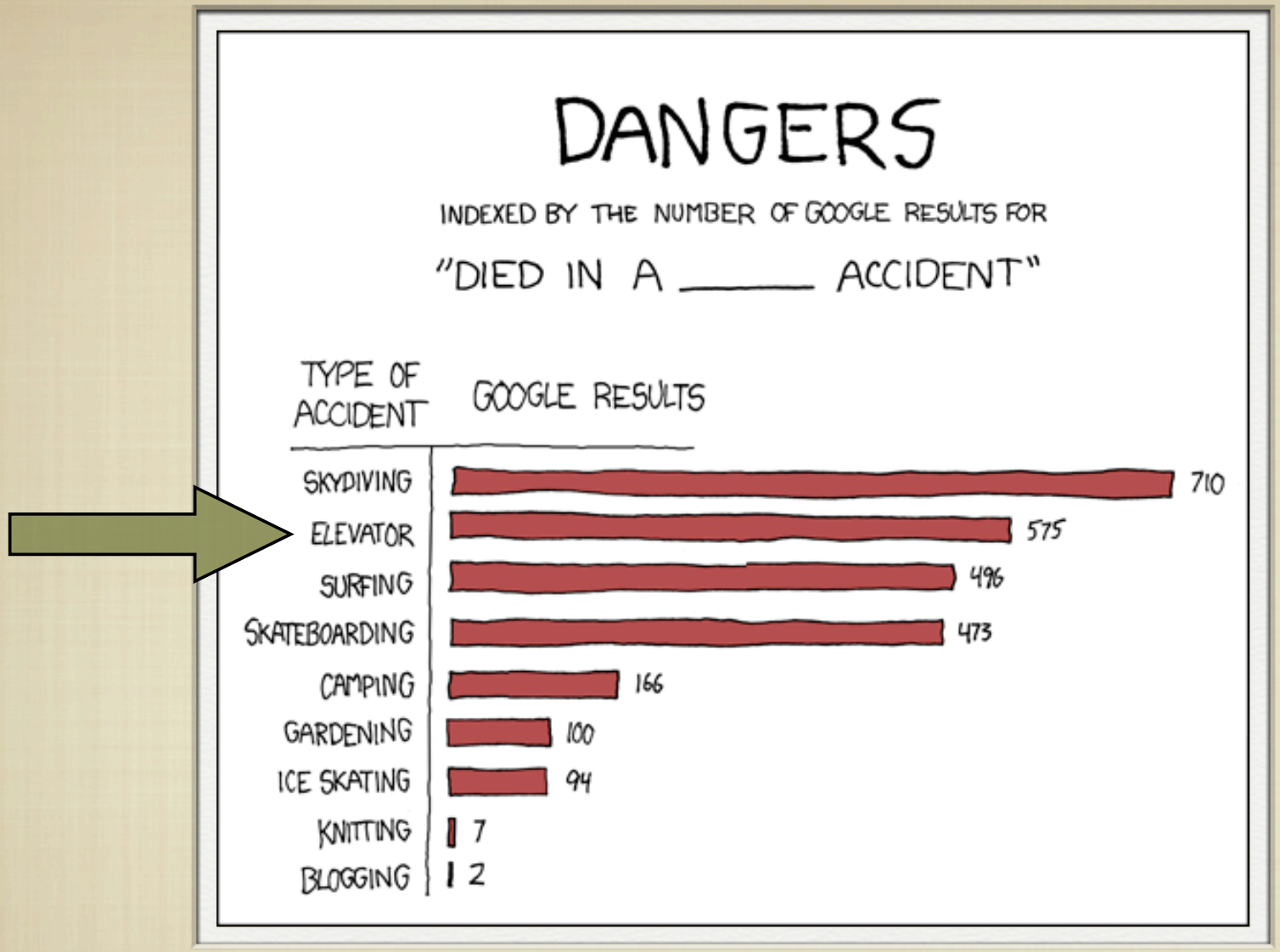
Alloy Modules, Integers, Analysis

Curt Clifton
Rose-Hulman Institute of Technology



Zero results:

'snake charming' and 'haberdashery'

http://www.bigelowaerospace.com

Modules

Module System

- Declare module: **module** pathname
- Use module: **open** pathname
- Alloy uses actual location of main module to guess location of opened modules
  - Suppose this file is in C:\Desktop\main.als:
    - **module** filesystem/main
      open filesystem/debug
      open filesystem/library/dirmodel
      ...

Avoiding Name Clashes

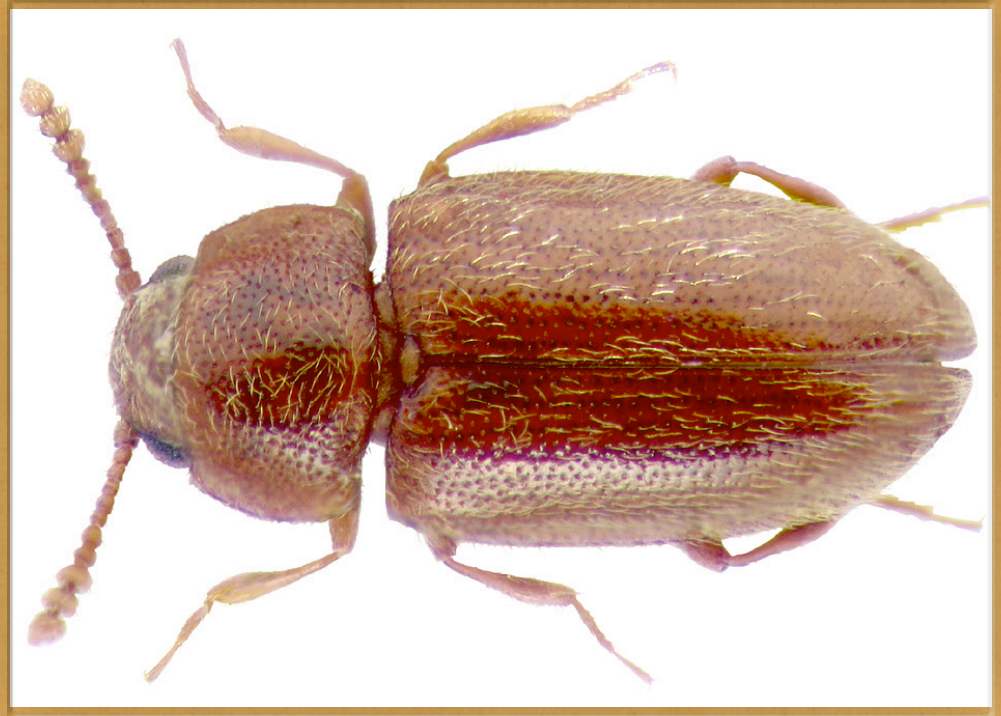
- Use aliases:
  - **open** library/graphs **as** G
    ...
    pred Acyclic[] { G/Acyclic[parents] }

Parametric Modules

- Like generic types in C++
- Example declaration:
  - **module** library/tree[nodeType]

    pred isTree[r: nodeType -> nodeType] { ... }
    ...
- Example use:
  - **open** library/tree[Object]

Cryptophilus integer Heer



http://www.zin.ru/animalia/Coleoptera/eng/cryintkm.htm

Alloy Integers

Alloy Integers

- Has Int atoms and int values
- Like Java's Integer wrapper class vs. int type
- int → Int
  - Alloy: Int[4]
  - Java: new Integer(4)
- Int → int
  - Alloy: int[e]
  - Java: int s = 0; for(Integer i : e) { s += i.intValue(); }

?!?

Integer Example

```
// Weighted Graph
sig Node {
    adj: Int lone -> Node
}
fact {
    all n: Node |
        let selfEdges = n.adj.n |
            some selfEdges => int[selfEdges] = 0
}
run {} for exactly 3 Node
```

Why wait so long?

- Not actually very useful
- "If you think you need them, think again."
- What properties do you really need:
  - Unique IDs? Just use atoms and a fact to make them disjoint
  - Ordering? Use util/ordering to impose order on atoms

Booleans?

- Avoid them! Conflict with set logic and make constraints too "wordy".

- Use sub-signatures:

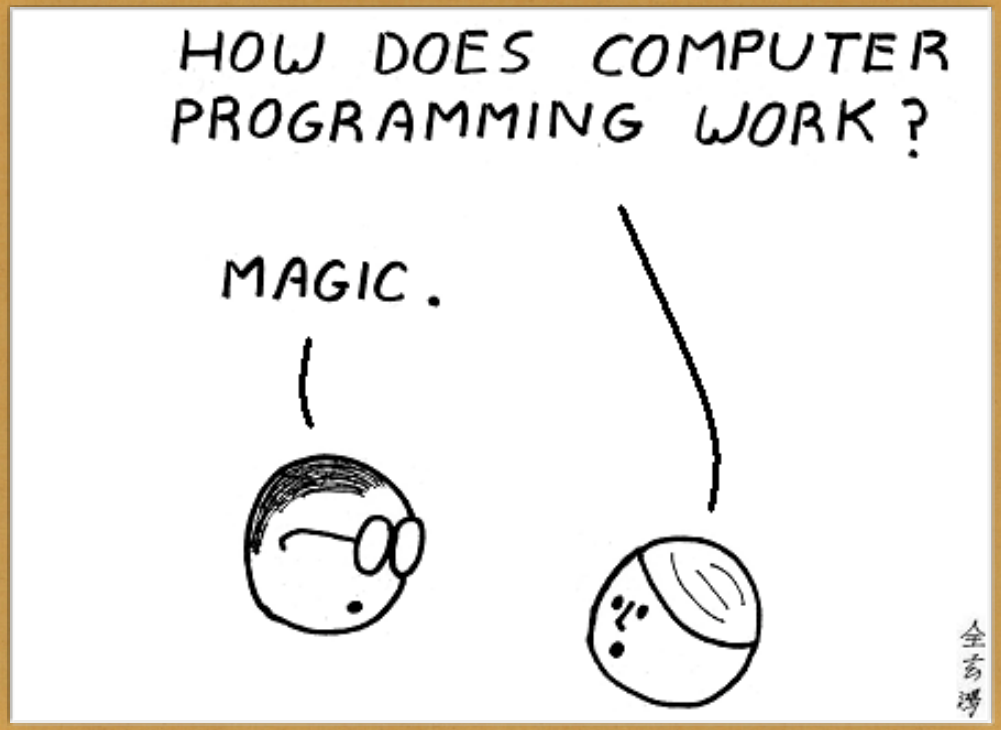
```
// BAD!
sig Lift { ...
      doorOpen: Time -> Boolean
}
```

```
// Good!
abstract sig DoorState{}
one sig Open, Closed extends DoorState{}
sig Lift { ...
      door: Time -> DoorState
}
```

- Use options:

```
// BAD!
sig Lift { ...
      upArrowLit: Time -> Boolean
      downArrowLit: Time -> Boolean
}
```

```
// Good!
abstract sig Direction{}
one sig Up, Down extends Direction{}
sig Lift { ...
      arrowLit: Time -> lone Direction
}
```



Alloy Analysis

A peek under the hood

Equivalent Problems

- Checking assertions is equivalent to running predicates
- Find some assignment of relations to variables that makes constraints true
  - Predicates:
    sigs + fields + facts + **predicate constraints**
  - Assertions:
    sigs + fields + facts + **not(assertion constraints)**

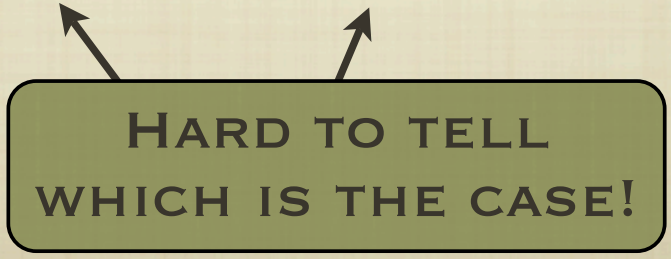
The Problem

- Alloy's logic is **undecidable**
  - Impossible to build a tool that can check every assertion
  - Must make some compromises

Any logic strong enough to model software systems is (probably) undecidable

Traditional Compromise

- Use automated theorem proving
  - Tries to build proof that assertion holds
  - Requires guidance from user
- Success? – assertion definitely holds
- Failure? – assertion may or may not hold



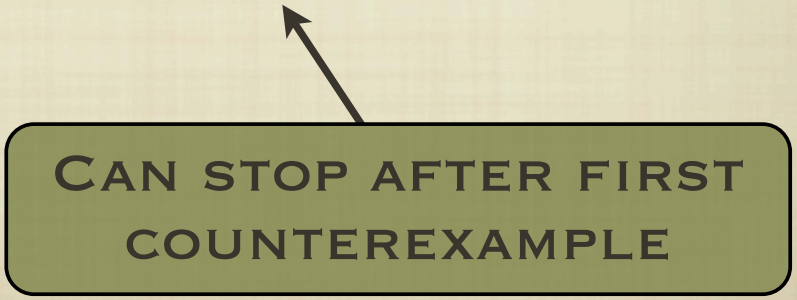
Q4

Alloy's Compromise

- Use **instance finding**
  - Looks for a "refutation" by checking against a huge set of test cases
- Success? – produces counterexample
- Failure? – assertion may hold, or there may be a larger counterexample than was checked

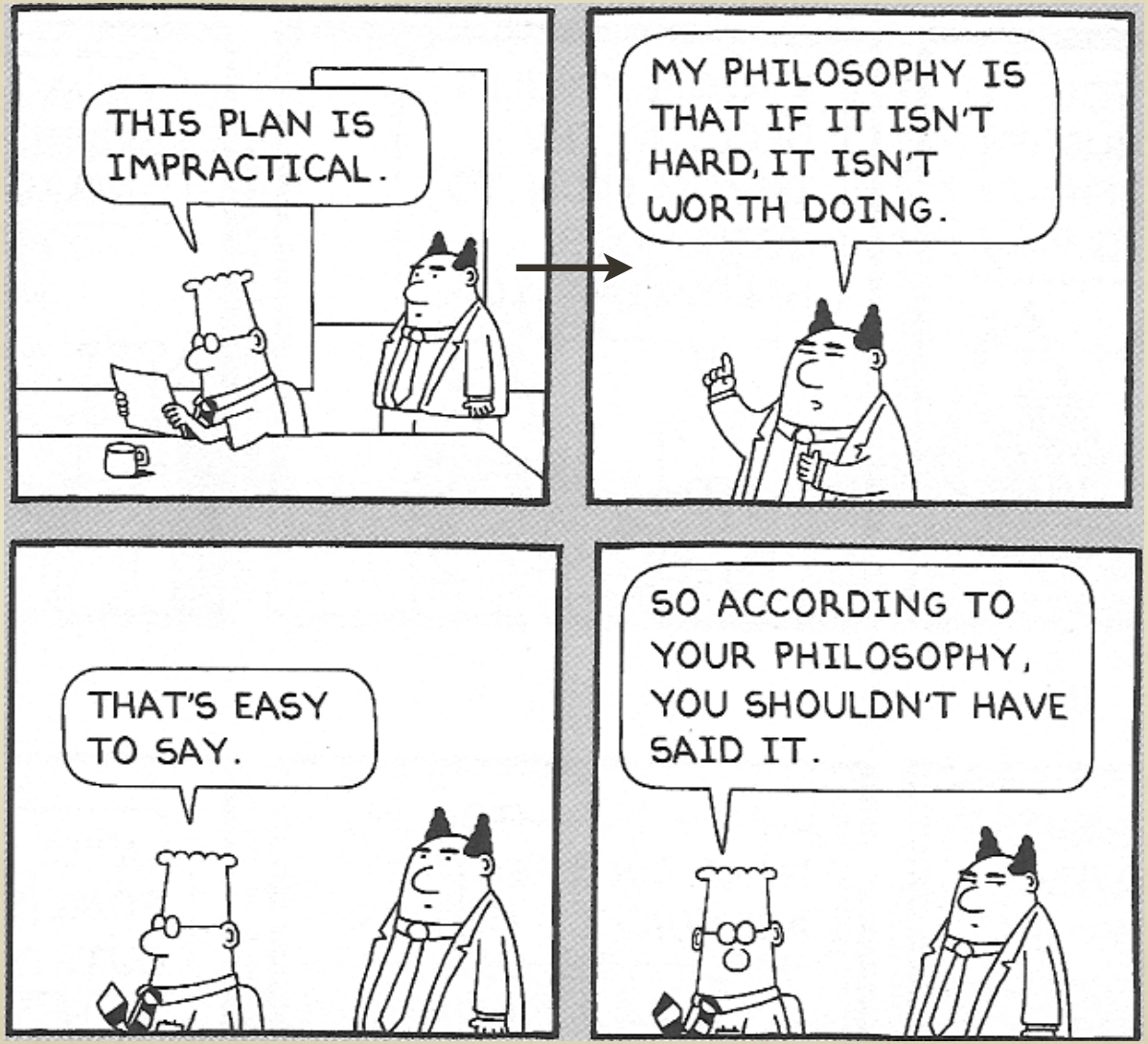
Why Instance Finding?

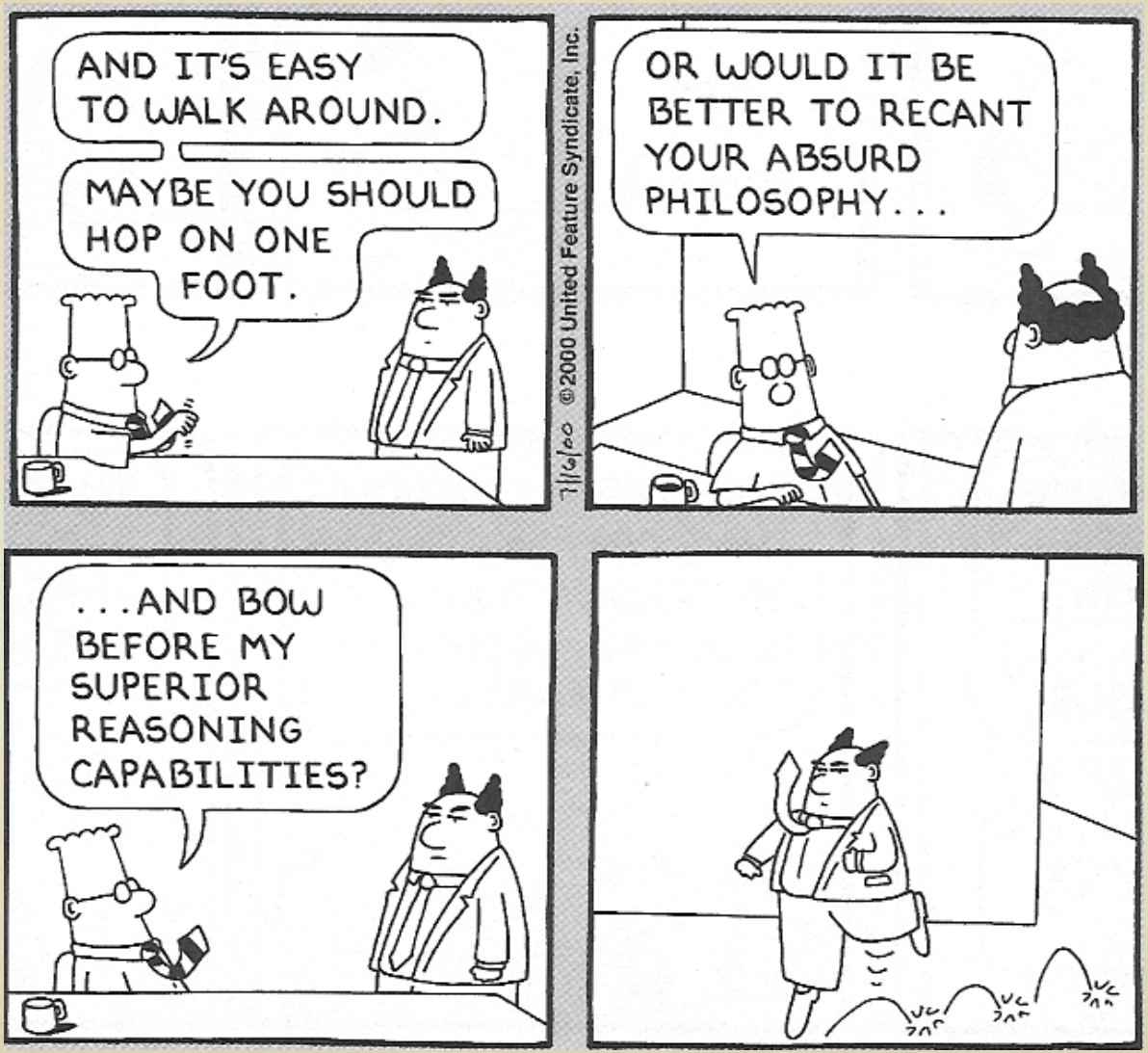
- During exploratory modeling most assertions will be invalid
- Specific counterexamples are helpful
- Invalid assertions can be found quickly



Q5

Cartoon of the Day




AND IT'S EASY TO WALK AROUND.
MAYBE YOU SHOULD HOP ON ONE FOOT.
OR WOULD IT BE BETTER TO RECANT YOUR ABSURD PHILOSOPHY...
©2000 United Feature Syndicate, Inc.
...AND BOW BEFORE MY SUPERIOR REASONING CAPABILITIES?

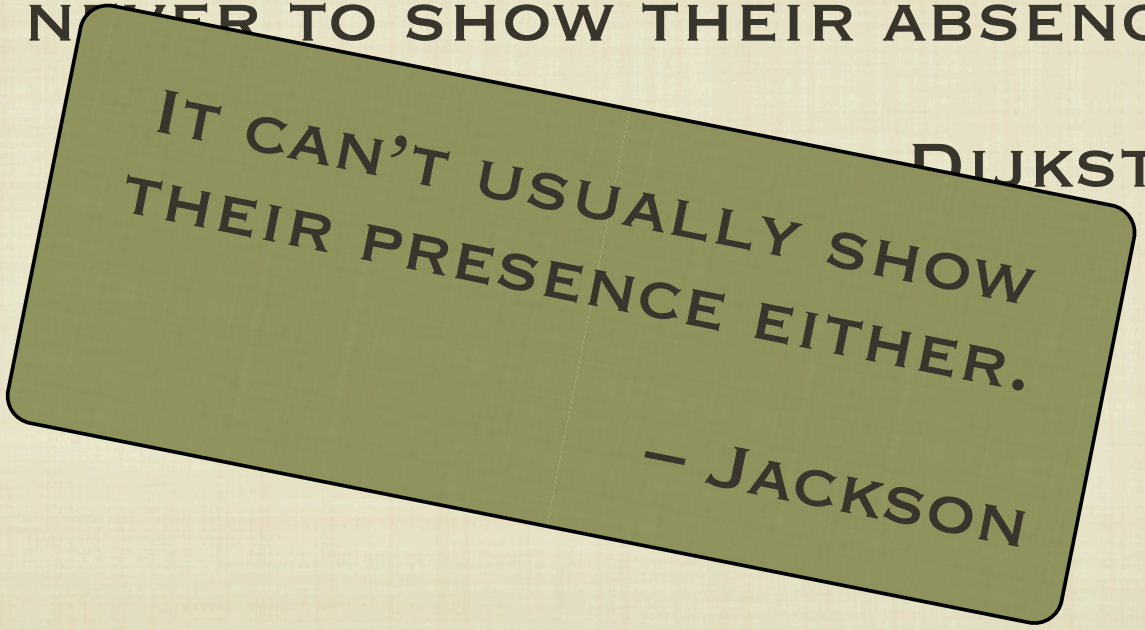
Tractable Instance Finding

- Limit scope of each signature to control size of search space
- Still handles **huge** number of test cases
- My P1 solution:
  - 1 lift, 5 floors, 25 moments in time, 2 directions, 2 door states:
    - current floor relation: 25 possibilities
    - requests relation: 25×2^5 possibilities
    - direction indicator relation: 25×3 possibilities
    - planned direction relation: 25×3 possibilities
    - doors relation: 25×2 possibilities

More than 5.6 billion instances checked!

Program testing can be used
to show the presence of bugs,
but never to show their absence.

Dijkstra



The Small Scope Hypothesis

Most bugs have small counterexamples

Q6,7