analyzing objects

Daniel Jackson
(joint work with Mandana Vaziri & Allison Waingold)
MIT Lab for Computer Science
NDIST · December 5, 2001

problem

want a simple framework

- for specifying & analyzing OO programs
- at the design level (like types)
- automatic (like model checking)

desiderata

- handles tricky cases
- helpful reports (eg, counterexamples)
- unlikely to miss errors
- modular: no whole-program assumption

what goes wrong?

```
// extends the map m with arbitrary key/val pairs
// taken from ks and vs, until one of ks and vs is empty

static void zippish (Map m, Set ks, Set vs) {
    Iterator ki = ks.iterator ();
    Iterator vi = vs.iterator ();
    while (ki.hasNext() && vi.hasNext()) {
        m.put (ki.next(), vi.next());
        ki.remove ();
        vi.remove ();
        }
    }
```

failure conditions

```
static void zippish (Map m, Set ks, Set vs) {
    Iterator ki = keys.iterator ();
    Iterator vi = vs.iterator ();
    while (ki.hasNext() && vi.hasNext()) {
        m.put (ki.next(), vi.next());
        ki.remove ();
        vi.remove ();
        }
    }
```

ks contains a key of m
- map is overwritten

ks, vs or m is null
- null pointer exception

ks or vs is a key of m
- rep invariant of m broken, unpredictable effect

ks and vs aliased
ks is a view of vs, or vice versa
ks or vs is a view of m
- comodification exception thrown

roadmap

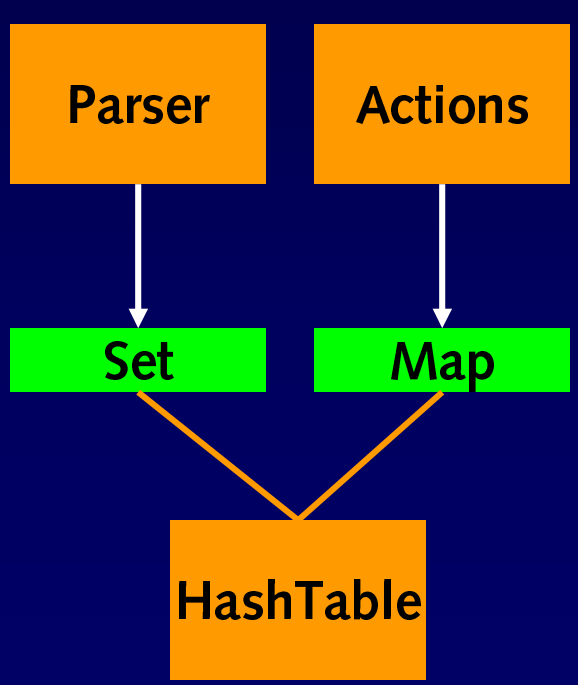
why views?
modelling the heap & mutation
modelling execution
modelling views

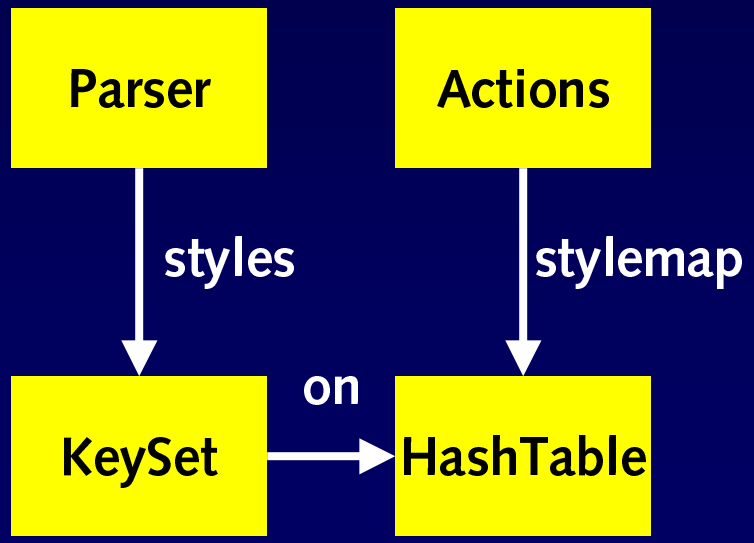
underlying theme
- simple first-order model of execution
- use constraint solver to find counterexample traces

why views?

decouples client from irrelevant aspect of datatype
- by weakening specification

interface Map {… Set keySet (); … }



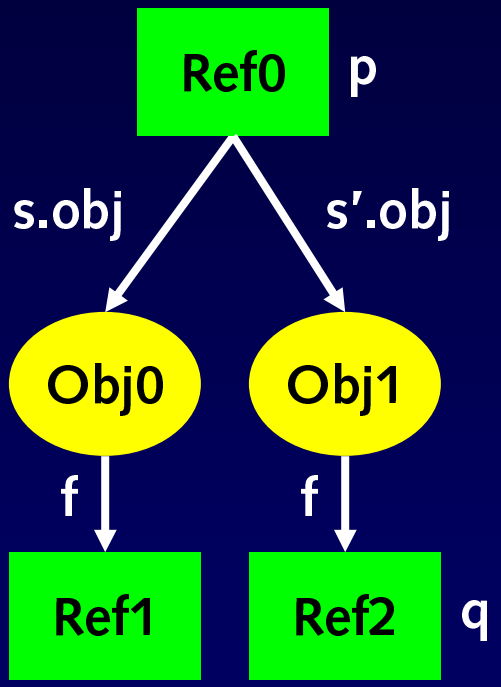


modelling the heap & mutation

two kinds of atoms

- references -- slots that hold objects
- objects -- values that go in slots

```
sig Ref {}
sig Obj {}
sig State {
    refs: set Ref,
    obj: refs ->? Obj
    }
sig X extends Obj {f: Ref}
fun op (s, s': State, p, q: Ref) {
    p.(s'.obj).f = q
    }
```



frame conditions

```
sig Set extends Obj {elts: set Ref}
sig SetRef extends Ref {}
fact {SetRef.(State.obj) in Set}

fun modifies (s, s': State, rs: set Ref) {
    all r: s.refs - rs | r.(s.obj) = r.(s'.obj)
    }

fun add (this: SetRef, s, s': State, e: Ref) {
    this.(s'.obj).elts = this.(s.obj).elts + e
    modifies (s, s', this)
    }
```

modelling execution

each statement

- modelled as a parameterized formula
- relating pre- and post-state
- instantiated with states at each point

```
BODY =   stmt0 (s, s1, …)   &&
         stmt1 (s1, s2, …)  &&
         …                  &&
         stmtN (sN, s', …)
```

every state satisfies a condition

- all s, s1, …, s': State | Pre(s) && BODY => P(sK)

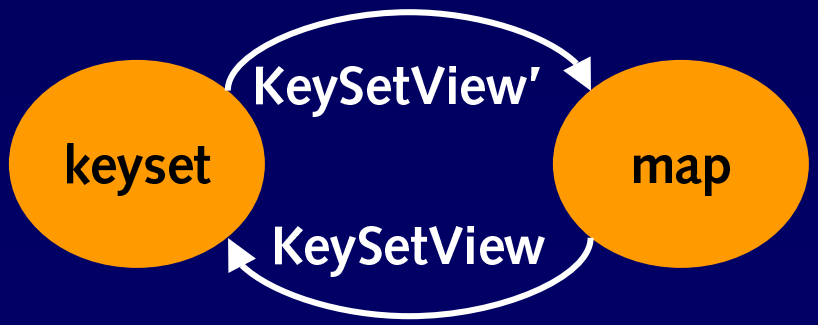
method satisfies spec

- all s, s1, …, s': State | Pre(s) && BODY => Post(s')

modelling views

views are not projections
- map is also view of keyset
- say v is view of b when change to b is propagated to v

not all views are bidirectional
- keyset/map is two-way
- iterator/set is one-way
- model bidirectional case with two views
- modification of view which does not back another view invalidates it



modelling views (generic)

state holds view relationships

```
sig State {
    refs: set Ref,
    obj: refs ->! Object,
    views: ViewType -> refs -> refs
    }
```

frame condition propagates changes to view

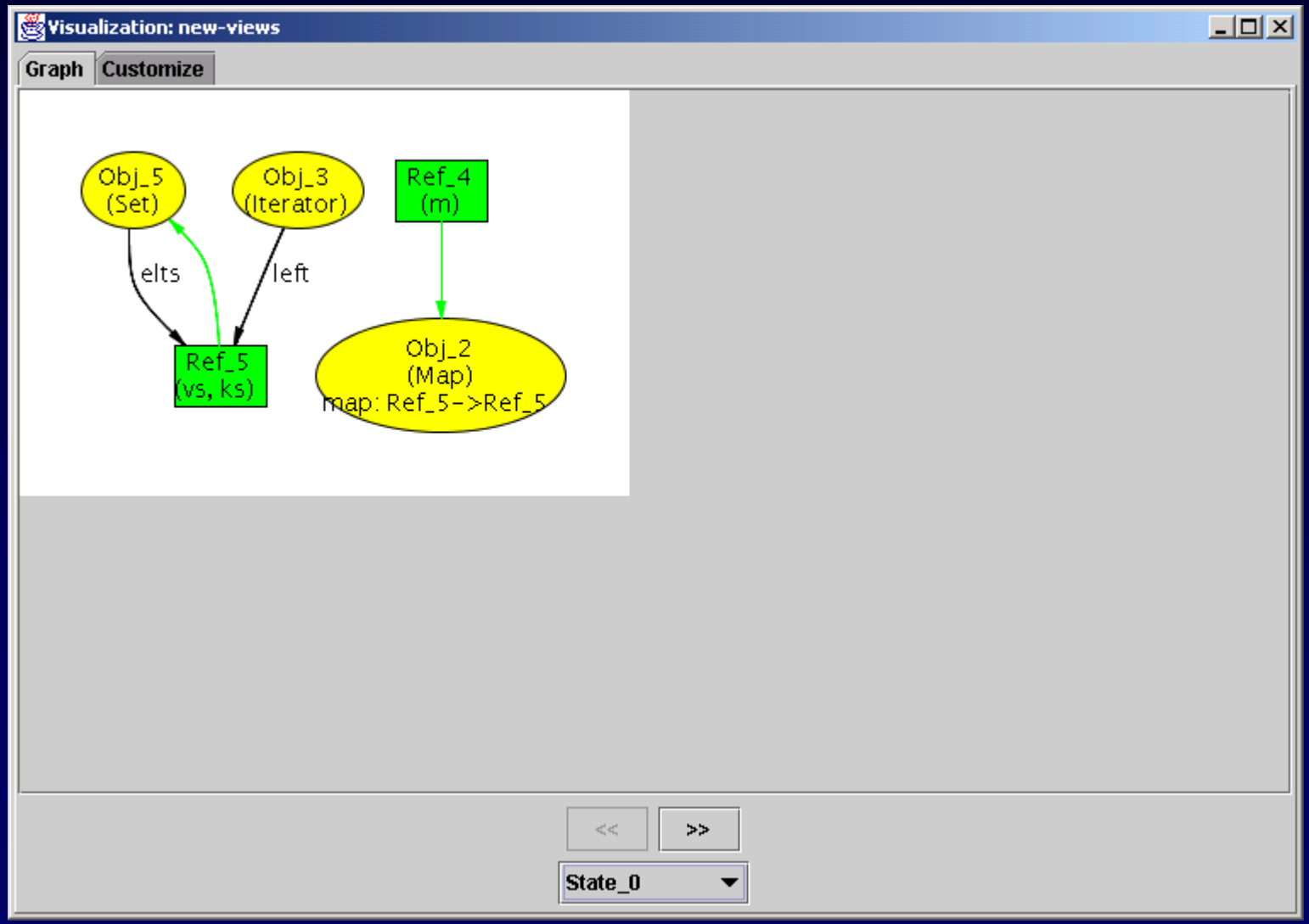
```
fun modifies (s, s': State, rs: set Ref) {
    let vr = ViewType.(s.views), mods = rs.*vr {
      all r: s.refs - mods | r.(s.obj) = r.(s'.obj)
      all b: mods, v: s.refs, t: ViewType |
          b->v in t.(s.views) =>
              viewFrame (t, v.(s.obj), v.(s'.obj), b.(s'.obj))
      }}
```

modelling views (view-specific)

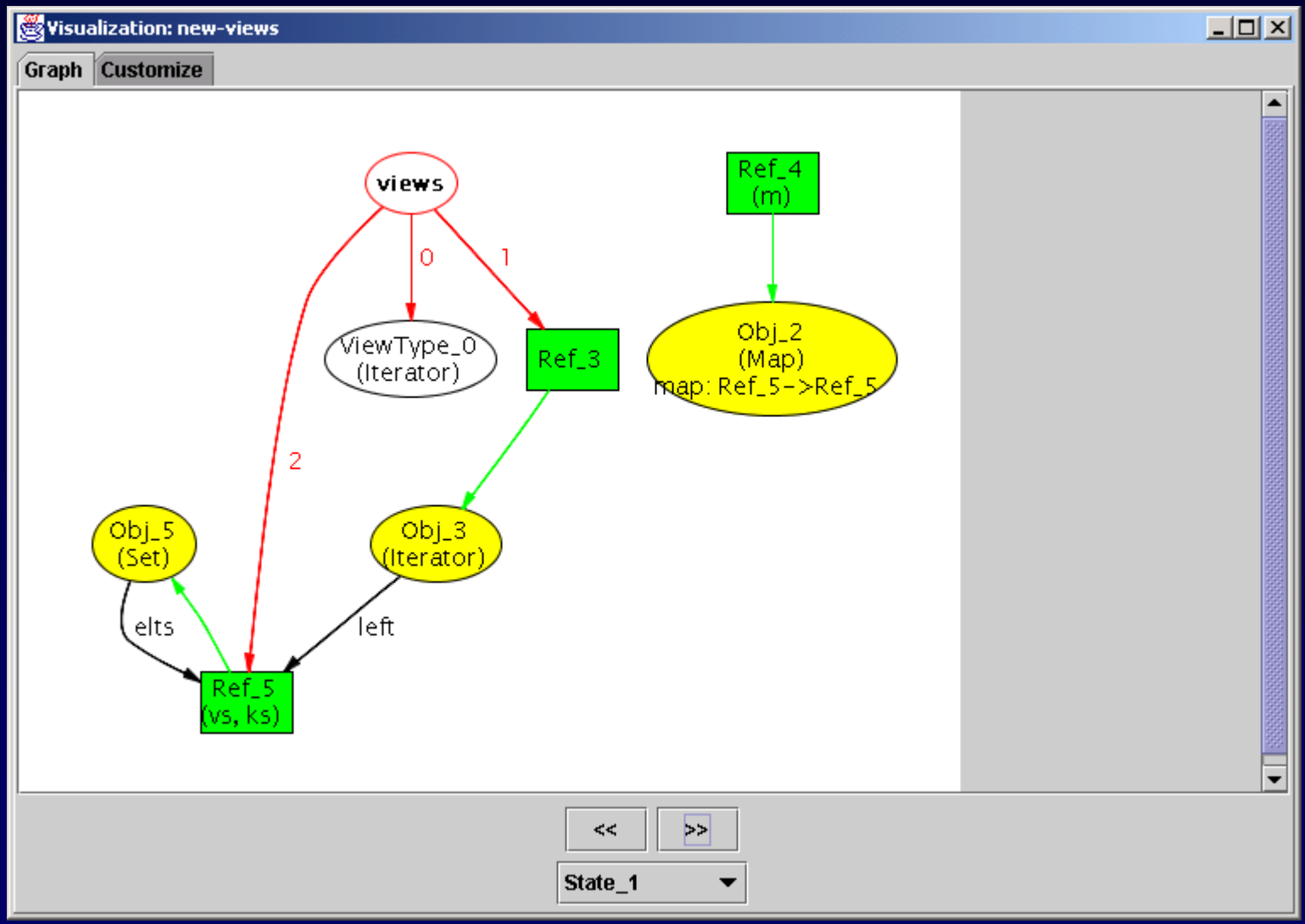
```
disj sig KeySetView, KeySetView', IteratorView extends ViewType {}

fun viewFrame (t: ViewType, v, v', b': Object) {
    t in KeySetView => v'.elts = dom (b'.map)
    t in KeySetView' => b'.elts = dom (v'.map)
    t in KeySetView' =>
       domRestrict (b'.elts, v.map) = domRestrict (b'.elts, v'.map)
    t in IteratorView => v'.elts = b'.left + b'.done
    }
```

demo: a counterexample generated

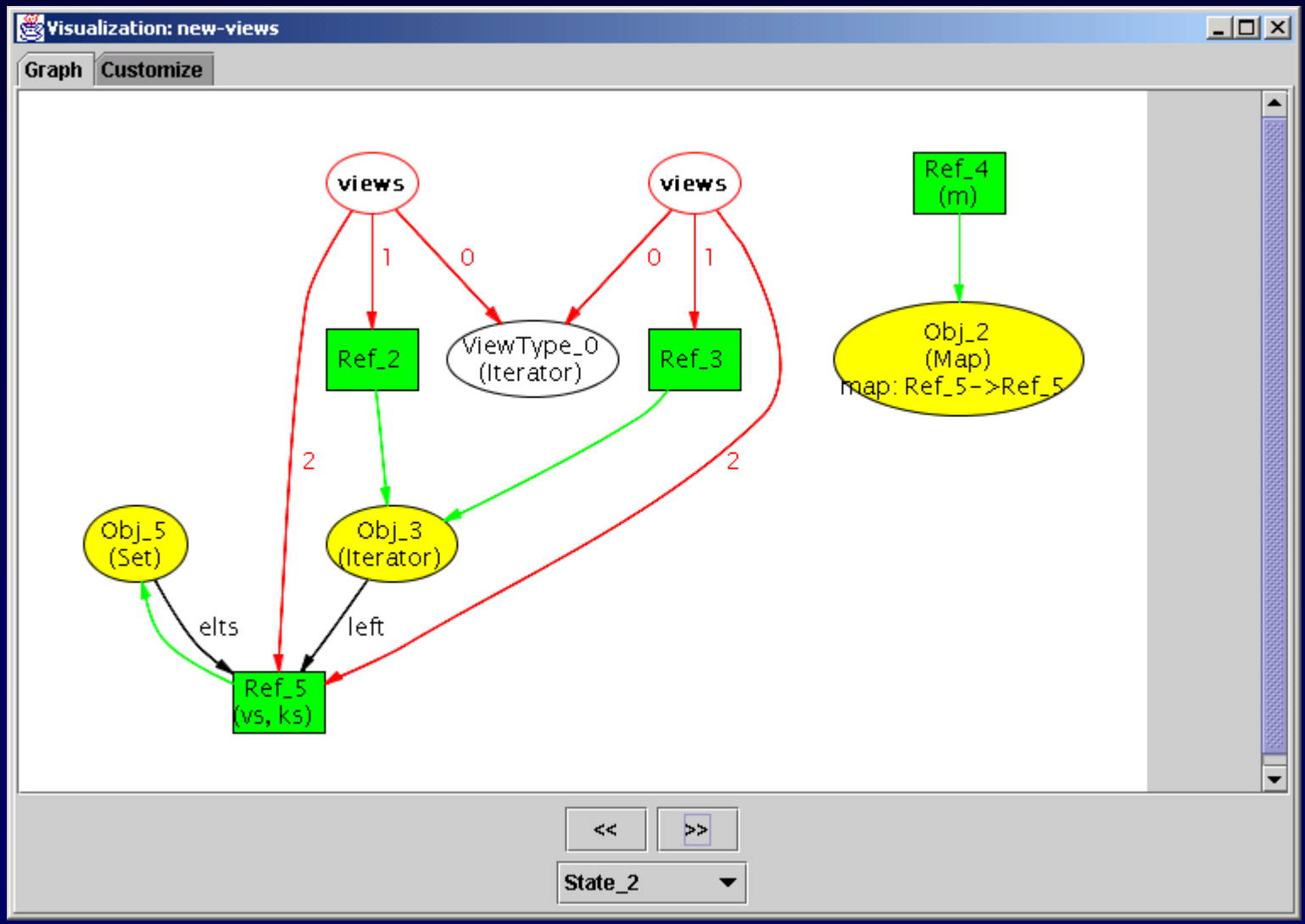
pre-state: ks and vs aliased



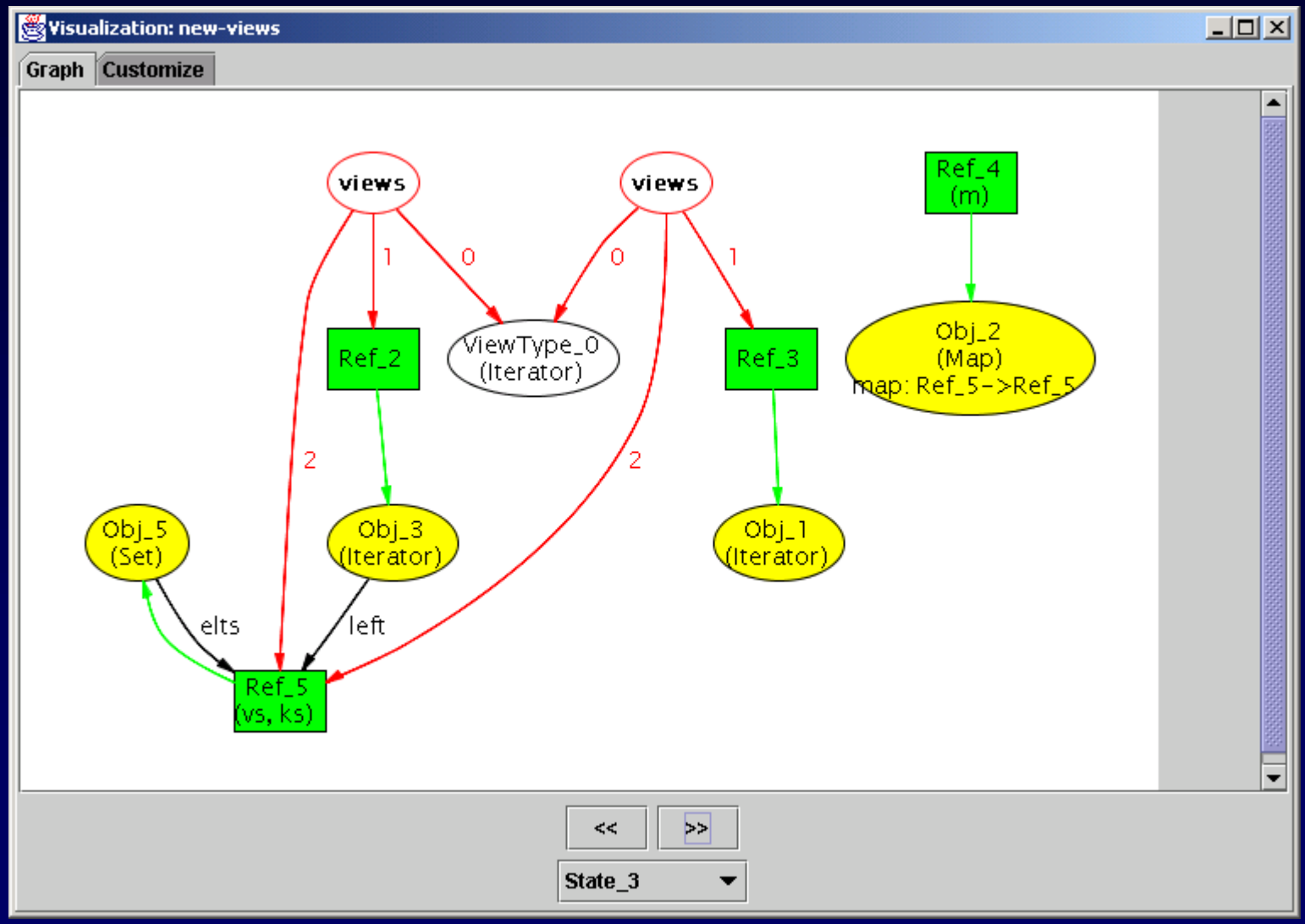
after: ki = ks.iterator ()



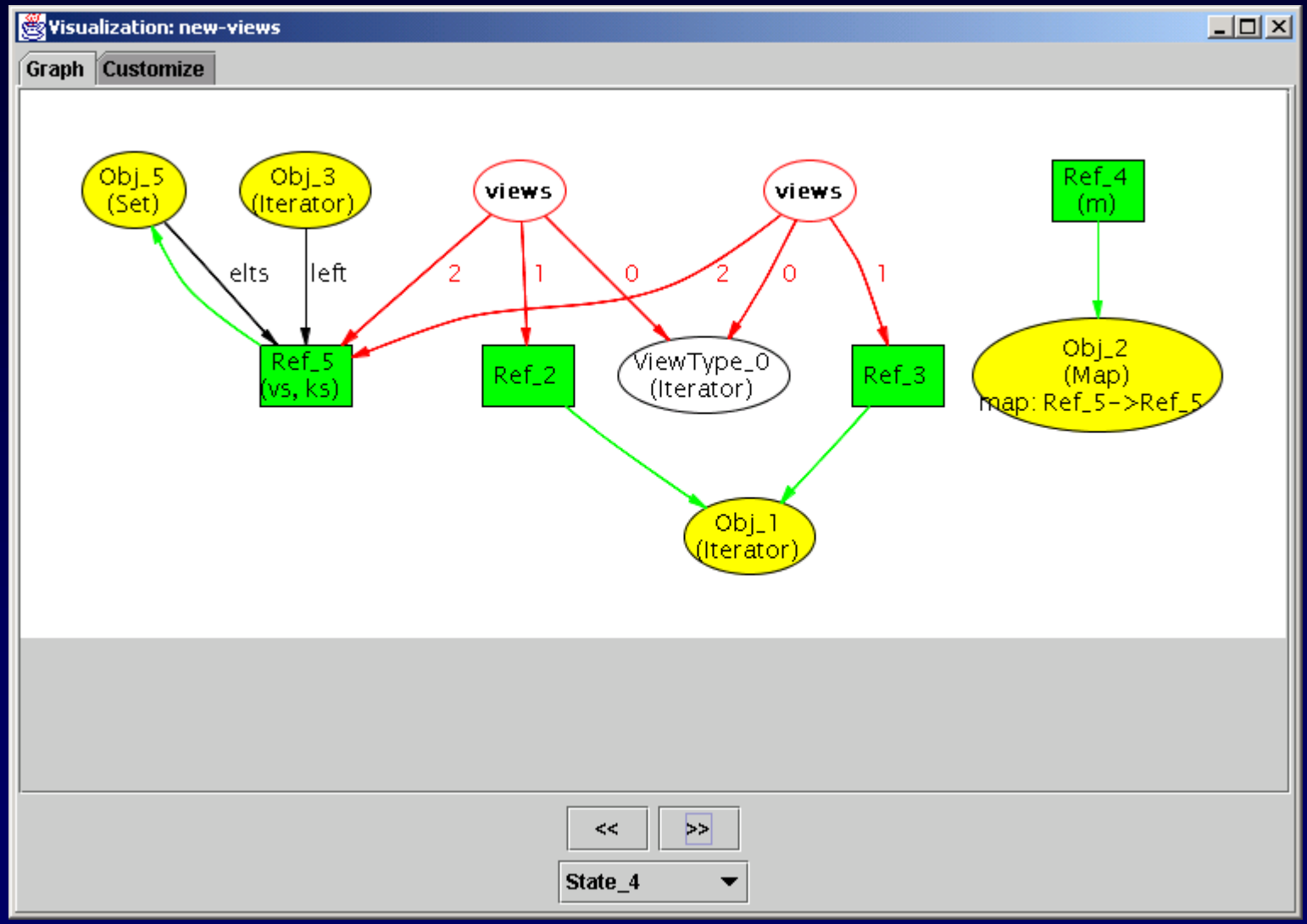
after: vi = vs.iterator ()



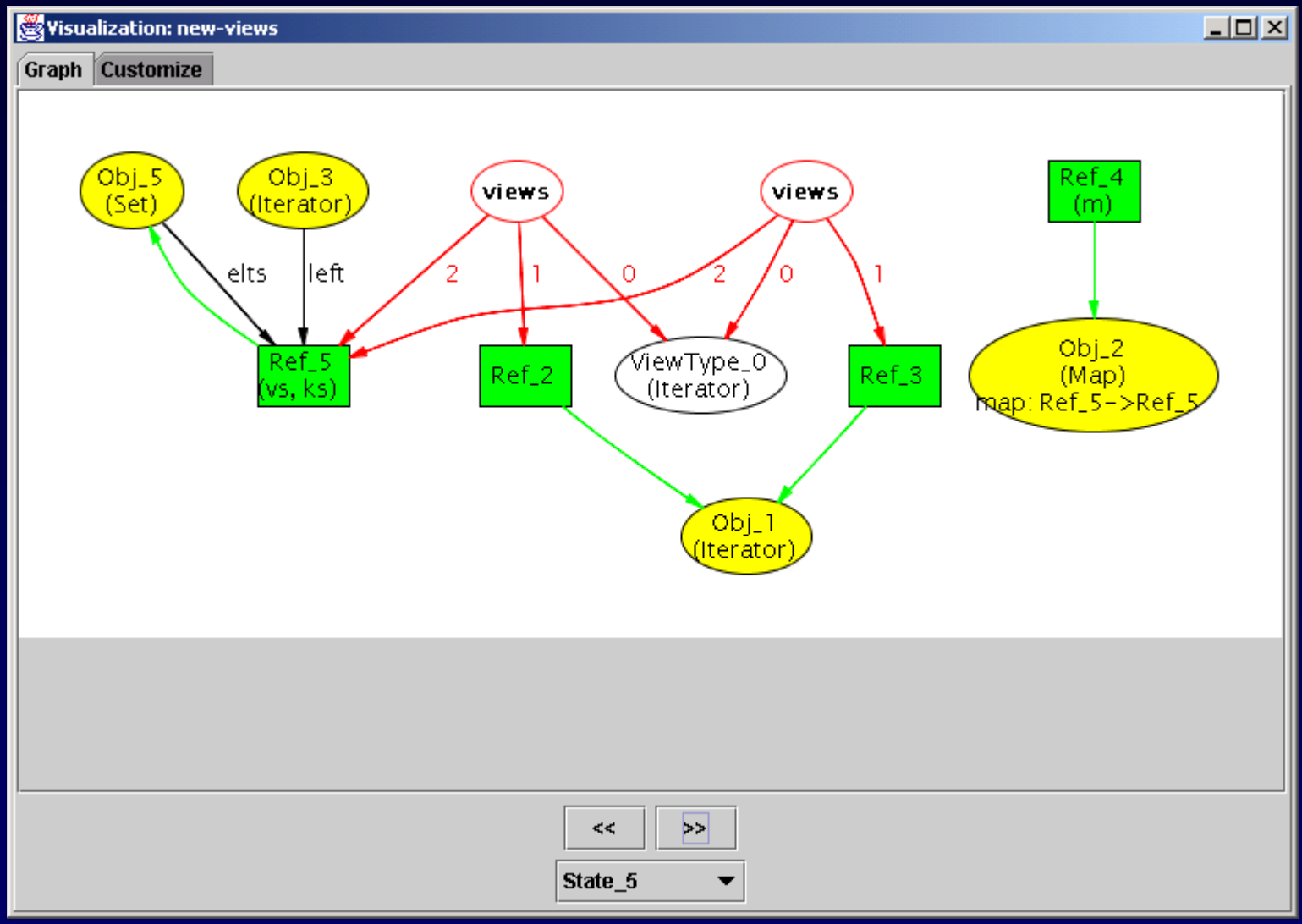
after: k = ki.next ()



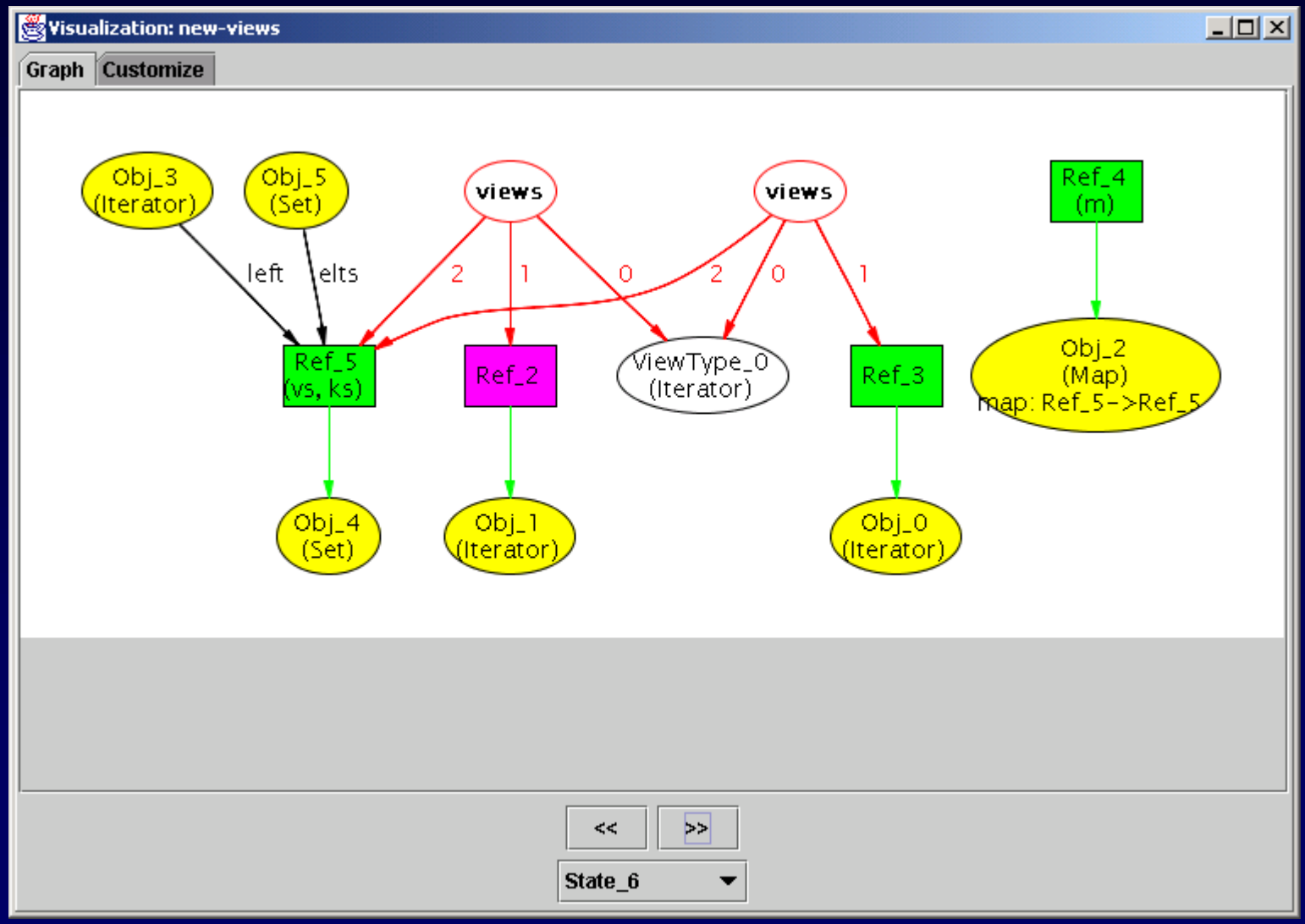
after: v = vi.next ()



after: m.put (k,v)



after: ki.remove() *Ref_2 gets invalidated*



conclusions

scenario
- write specs for API's (and perhaps for program)
- extract relevant code skeleton
- use constraint solver to find bugs

unlike most other analyses
- doesn't require whole program
- can find subtle errors

but …
- will analysis scale enough?
- will specs still be too much work?
- will false alarms become a problem?