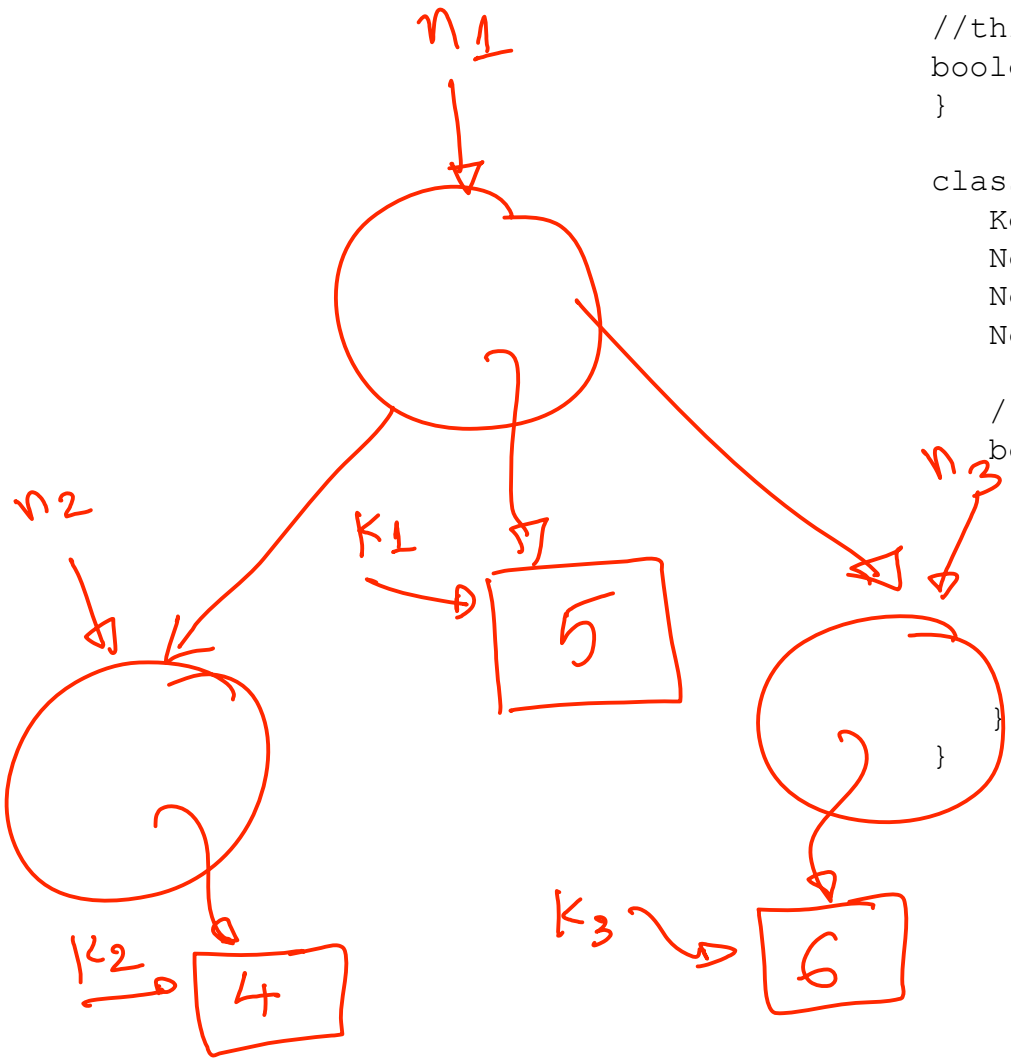
Runtime Monitoring of Object Invariants with Guarantees

Sriram Rajamani, MSR India

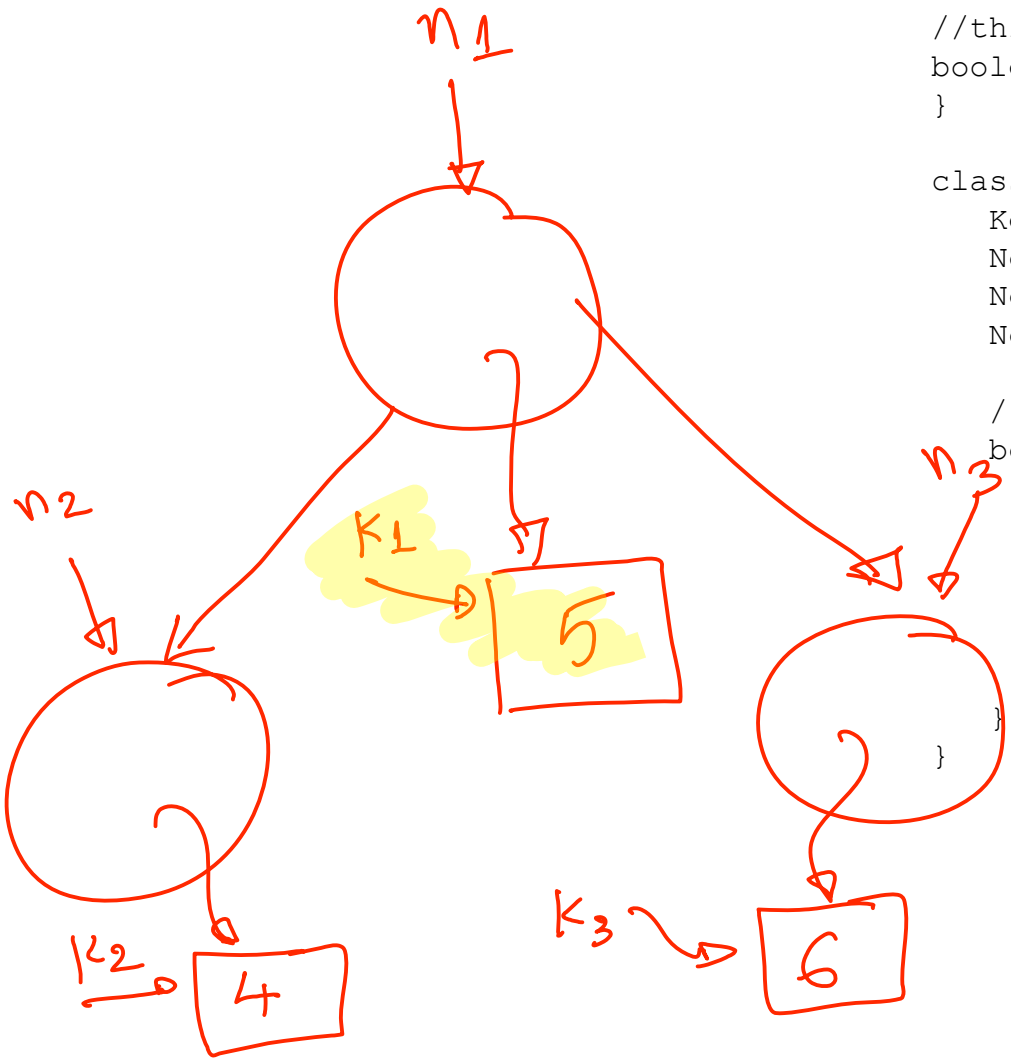
(joint work with Madhu Gopinathan, Indian Institute of Science)

Object invariants



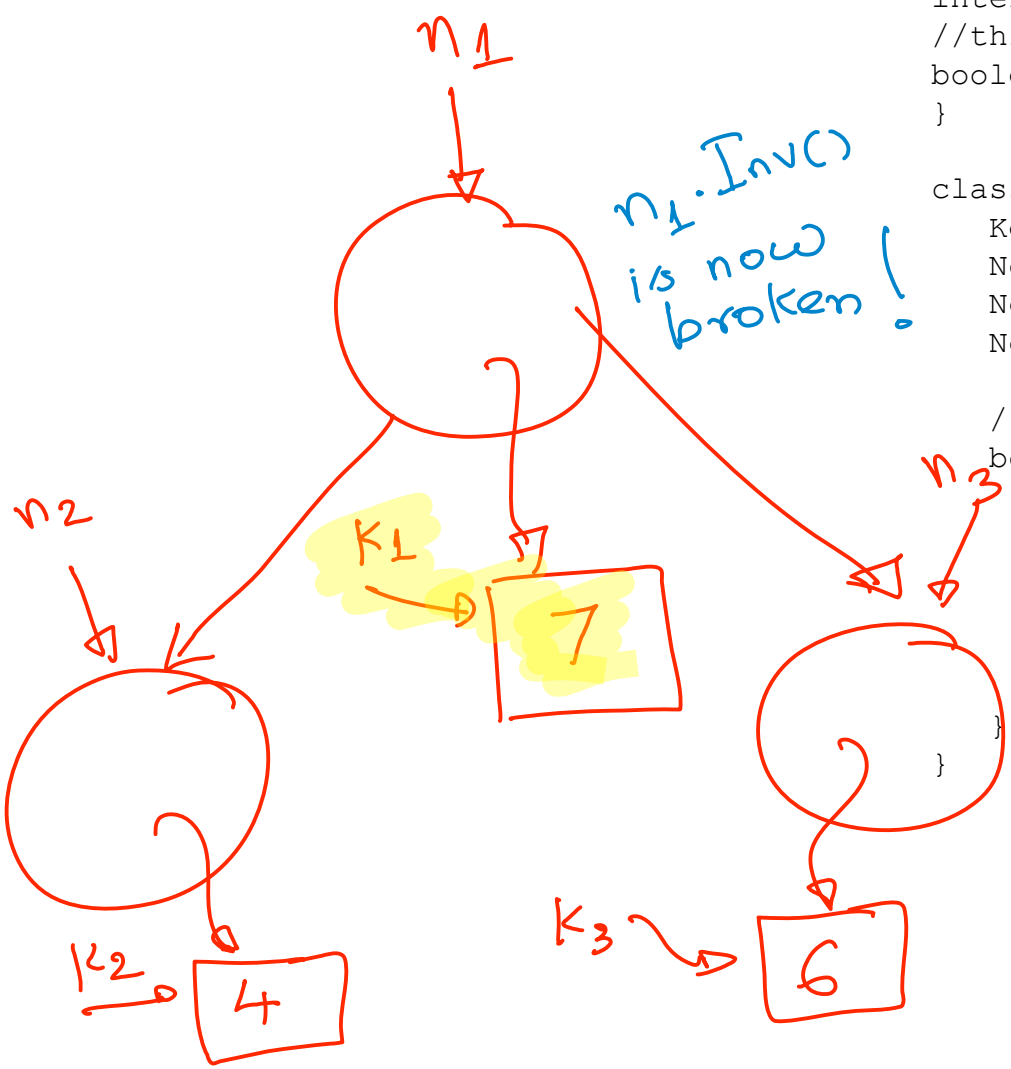

```
interface Key{
//this < k?
boolean Less(Key k);
}

class Node {
   Key k;
   Node left;
   Node right;
   Node(Key key) { k = key };

   //object invariant of this node
   boolean Inv(){
         return((left==null ||
                  left.k.Less(k) && left.Inv())
                &&
                 (right==null ||
                   k.Less(right.k)&&
                                right.Inv()));
   }
}
```

Object invariants



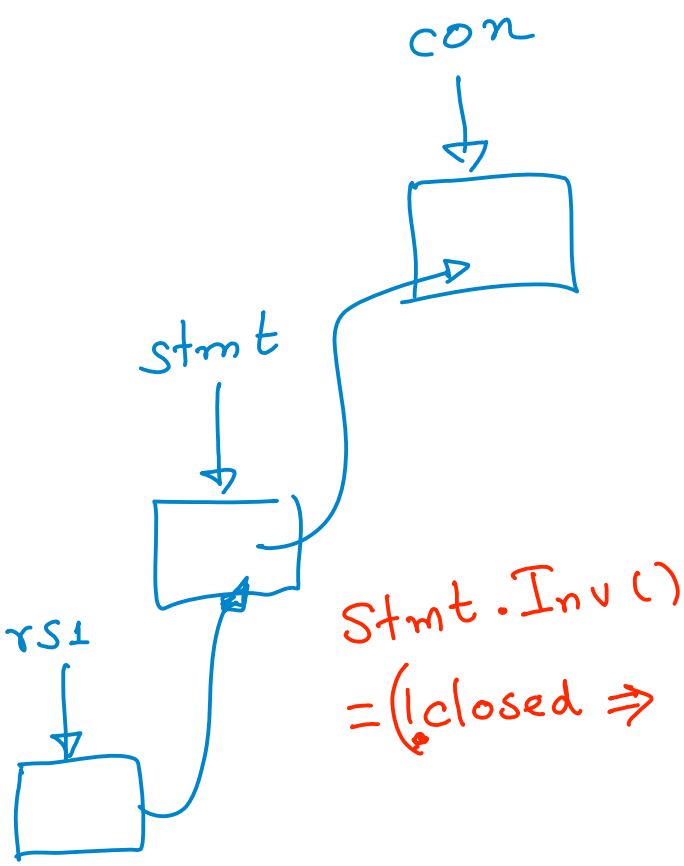

```
interface Key{
//this < k?
boolean Less(Key k);
}

class Node {
   Key k;
   Node left;
   Node right;
   Node(Key key) { k = key };

   //object invariant of this node
   boolean Inv(){
         return((left==null ||
                  left.k.Less(k) && left.Inv())
                &&
                 (right==null ||
                   k.Less(right.k)&&
                                right.Inv()));
   }
}
```

Object invariants




```
interface Key{
//this < k?
boolean Less(Key k);
}

class Node {
   Key k;
   Node left;
   Node right;
   Node(Key key) { k = key };

   //object invariant of this node
   boolean Inv(){
         return((left==null ||
                  left.k.Less(k) && left.Inv())
                &&
                 (right==null ||
                  k.Less(right.k)&&
                                right.Inv()));
   }
}
```

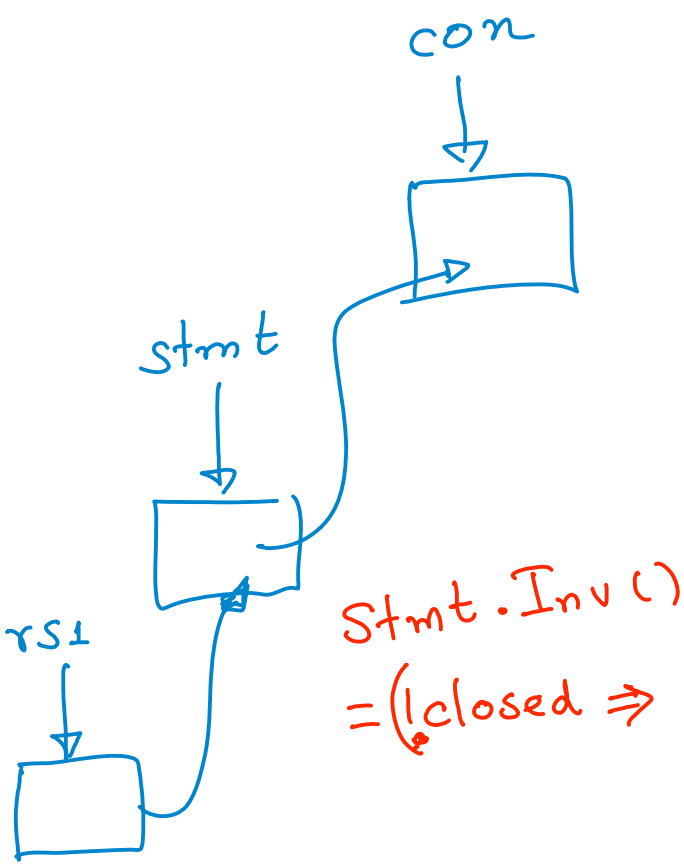
Another example from JDBC




```
Connection con =
   DriverManager.getConnection(..);
Statement stmt = con.createStatement();
ResultSet rs1 =  stmt.executeQuery("SELECT
                  EMPNOFROM EMPLOYEE");
..
con.close();
..

ResultSet rs2 = stmt.executeQuery("SELECT
                EMPNAME FROM EMPLOYEE");
```

Stmt.Inv()
= (!closed ⇒ ! con.closed)

Another example from JDBC



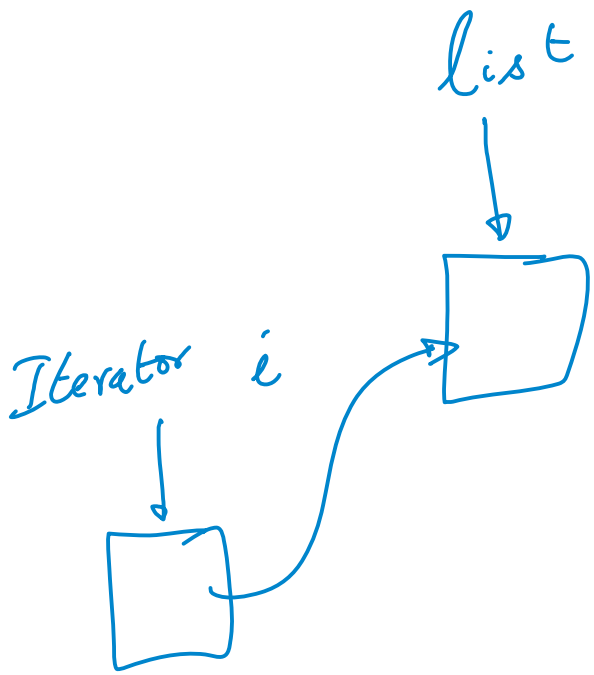

```
Connection con =
   DriverManager.getConnection(..);
Statement stmt = con.createStatement();
ResultSet rs1 =  stmt.executeQuery("SELECT
                  EMPNOFROM EMPLOYEE");
..
con.close();
..

ResultSet rs2 = stmt.executeQuery("SELECT
                EMPNAME FROM EMPLOYEE");
```

Stmt.Inv()
= (!closed ⇒ ! con.closed)

Violates Stmt.Inv()!

Another example - Iterators

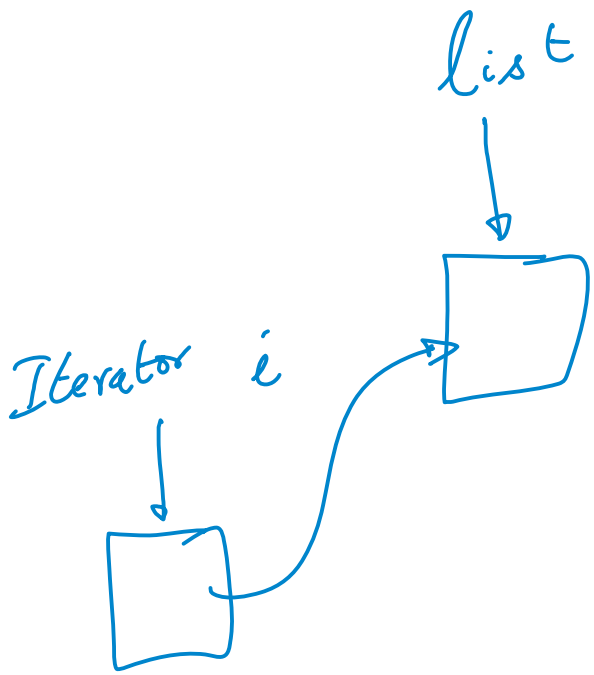
```
//list is of type ArrayList<Integer>
//with integers 1,2,3 added

for(Iterator<Integer> i = list.iterator();
   i.hasNext(); ) {
    int v = i.next();
    if(v == 1)
         list.remove(v);
    else
         System.out.println(v);
}
```



i.Inv() = (version == list.version)

Another example - Iterators

```
//list is of type ArrayList<Integer>
//with integers 1,2,3 added

for(Iterator<Integer> i = list.iterator();
    i.hasNext(); ) {
     int v = i.next();
     if(v == 1)
          list.remove(v);
     else
          System.out.println(v);
}
```



violates invariant of Iterator i

i.Inv() = (version == list.version)

Our goal

- A runtime scheme to monitor object invariants
- Guarantee to detect invariant violations when they happen

Two issues with checking invariants

- When does an object invariant hold?
  - When object o is in a “stable state”

- When object o’s invariant depends on p, what happens when p changes without o’s knowledge?

Reusable Monitor

like an interface but with state

```
role ObjWInv {
  boolean Inv();
  boolean inv;
}
```

auxeliary state added to all objects

$$\forall o \in ObjWInv \quad o.inv = true \Rightarrow o.Inv()$$

Reusable Monitor

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}
```

Keep track of other objects that depend on o

$$\forall o \in ObjWInv \quad o.inv = true \Rightarrow o.Inv()$$

Reusable Monitor

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}

Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}
```

called when o is created

$$\forall o \in ObjWInv \quad o.inv = true \Rightarrow o.Inv()$$

Reusable Monitor

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}

Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}
```

only way to set o.inv

```
CheckAndSetInv(ObjWInv o) {
  assert o.Inv();
  o.inv = true;
}
```

adding dependents

```
Add(ObjWInv o, ObjWInv p) {
  assert(o.inv = false);
  p.dependents.Add(o);
}
```

$$\forall o \in ObjWInv \quad o.inv = true \Rightarrow o.Inv()$$

Reusable Monitor

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}

Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}

CheckAndSetInv(ObjWInv o) {
  assert o.Inv();
  o.inv = true;
}


Add(ObjWInv o, ObjWInv p) {
  assert(o.inv = false);
  p.dependents.Add(o);
}
```

$$\forall o \in ObjWInv \quad o.inv = true \Rightarrow o.Inv()$$

Reusable Monitor

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}

Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}

CheckAndSetInv(ObjWInv o) {
  assert o.Inv();
  o.inv = true;
}

Add(ObjWInv o, ObjWInv p) {
  assert(o.inv = false);
  p.dependents.Add(o);
}
```

```
Start(ObjWInv o) {
  assert(o.inv = false);
  CheckAndSetInv(o);
}
```

called when we want to start monitoring the invariant

$$\forall o \in ObjWInv \quad o.inv = true \Rightarrow o.Inv()$$

Reusable Monitor

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}

Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}

CheckAndSetInv(ObjWInv o) {
  assert o.Inv();
  o.inv = true;
}

Add(ObjWInv o, ObjWInv p) {
  assert(o.inv = false);
  p.dependents.Add(o);
}
```

```
Start(ObjWInv o) {
  assert(o.inv = false);
  CheckAndSetInv(o);
}

Stop(ObjWInv o) {
  assert(o.inv = true);
  o.inv := false;
}
```

called when we want to stop monitoring the invariant

$$\forall o \in ObjWInv \quad o.inv = true \Rightarrow o.Inv()$$

Reusable Monitor

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}

Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}

CheckAndSetInv(ObjWInv o) {
  assert o.Inv();
  o.inv = true;
}

Add(ObjWInv o, ObjWInv p) {
  assert(o.inv = false);
  p.dependents.Add(o);
}
```

```
Start(ObjWInv o) {
  assert(o.inv = false);
  CheckAndSetInv(o);
}

Stop(ObjWInv o) {
  assert(o.inv = true);
  o.inv := false;
}

Validate(ObjWInv p) {
  for(o in p.dependents) {
    if(o.inv = true)
      CheckAndSetInv(o);
  }
}
```

called whenever p changes

$$\forall o \in ObjWInv \quad o.inv = true \Rightarrow o.Inv()$$

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}

Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}

CheckAndSetInv(ObjWInv o) {
  assert o.Inv();
  o.inv = true;
}

Add(ObjWInv o, ObjWInv p) {
  assert(o.inv = false);
  p.dependents.Add(o);
}

Start(ObjWInv o) {
  assert(o.inv = false);
  CheckAndSetInv(o);
}

Stop(ObjWInv o) {
  assert(o.inv = true);
  o.inv := false;
}

Validate(ObjWInv p) {
  for(o in p.dependents) {
    if(o.inv = true)
      CheckAndSetInv(o);
  }
}
```

Calling the monitor from the program

```
Connection con =
   DriverManager.getConnection(..);
Init(con); Start(con);

Statement stmt =
   con.createStatement();
Init(stmt);
Add(con, stmt);
Start(stmt);
 ...
Stop(stmt);
ResultSet  rs1 =
   stmt.executeQuery("SELECT..”);
Start(stmt);
 ...
con.close();
Validate(con);
```

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}

Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}

CheckAndSetInv(ObjWInv o) {
  assert o.Inv();
  o.inv = true;
}

Add(ObjWInv o, ObjWInv p) {
  assert(o.inv = false);
  p.dependents.Add(o);
}

Start(ObjWInv o) {
  assert(o.inv = false);
  CheckAndSetInv(o);
}

Stop(ObjWInv o) {
  assert(o.inv = true);
  o.inv := false;
}

Validate(ObjWInv p) {
  for(o in p.dependents) {
    if(o.inv = true)
      CheckAndSetInv(o);
  }
}
```

Calling the monitor from the program

```
Connection con =
   DriverManager.getConnection(..);
Init(con); Start(con);

Statement stmt =
   con.createStatement();
Init(stmt);
Add(con, stmt);
Start(stmt);
 ...
Stop(stmt);
ResultSet  rs1 =
   stmt.executeQuery("SELECT..");
Start(stmt);
 ...
con.close();
Validate(con);
```

What if programmer forgets this?

What if programmer forgets this?

Automated instrumentation

- When an object o is created, called Init(o)
- When a public method of o is entered, call Stop(o)
- When a public method of o is exited, call Start(o)

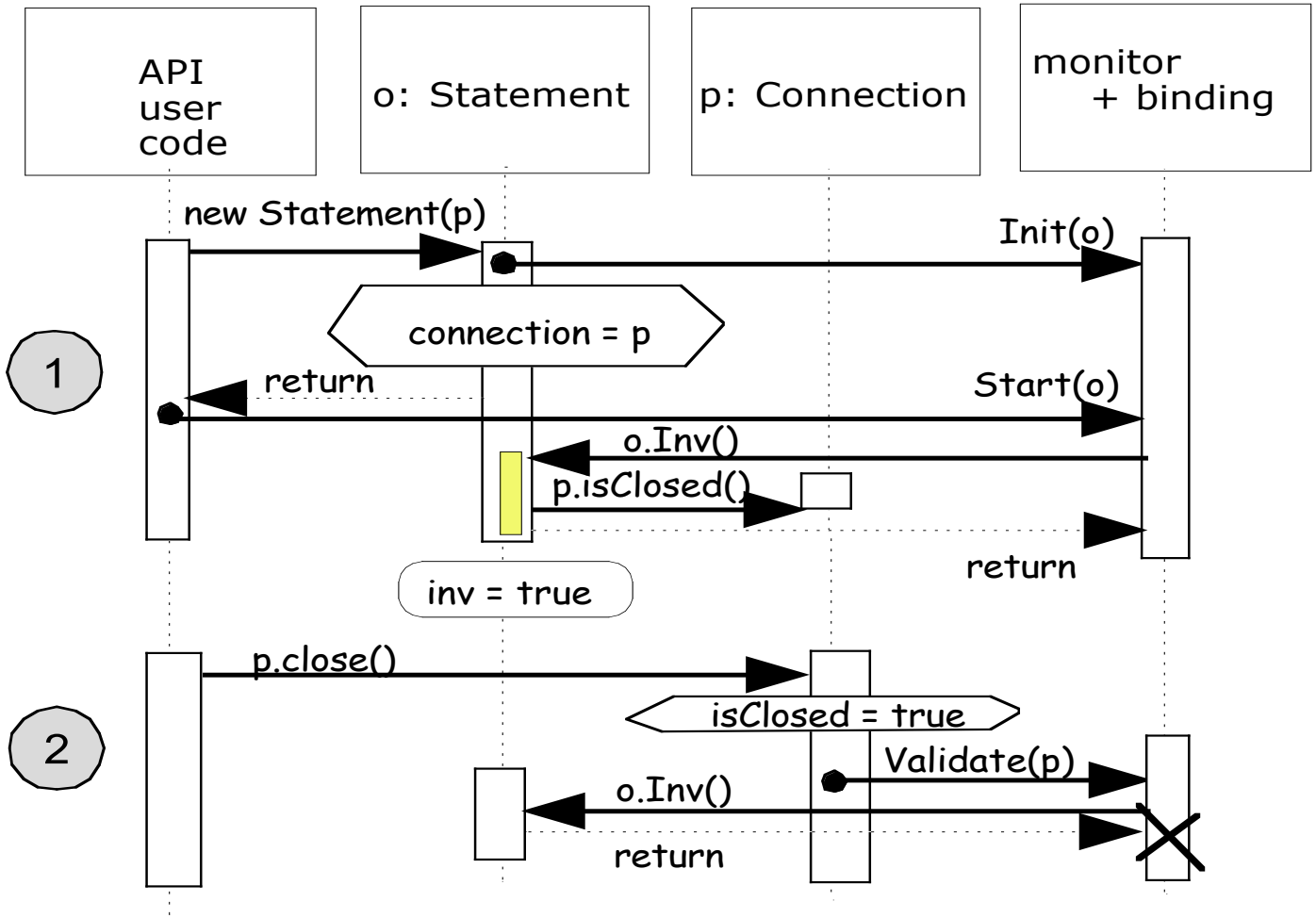
- Whenever o or dependents change, call "validate(o)" ! ← difficult to do!

```
role ObjWInv {
  boolean Inv();
  boolean inv;
  Set<ObjWInv> dependents;
}

Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}

CheckAndSetInv(ObjWInv o) {
  assert o.Inv();
  o.inv = true;
}

Add(ObjWInv o, ObjWInv p) {
  assert(o.inv = false);
  p.dependents.Add(o);
}

Start(ObjWInv o) {
  assert(o.inv = false);
  CheckAndSetInv(o);
}

Stop(ObjWInv o) {
  assert(o.inv = true);
  o.inv := false;
}

Validate(ObjWInv p) {
  for(o in p.dependents) {
    if(o.inv = true)
      CheckAndSetInv(o);
  }
}
```

Automatic Dependency Tracking

- Compute a relation D such that (o,p,f) in D iff the object invariant of o depends on the value of the field p.f
  - Can be done by AOP by monitoring all accesses during execution of o.Inv()
- Invoke Validate(p) whenever p.f changes.

Example


API
user
code
o: Statement
p: Connection
monitor
+ binding
new Statement(p)
Init(o)
connection = p
1
return
Start(o)
o.Inv()
p.isClosed()
return
inv = true
p.close()
isClosed = true
2
Validate(p)
o.Inv()
return

Correctness

Let r be any run of program P composed with the monitor using binding B. Suppose r does not have any assertion violations. Then, the following holds in all states of r:

$$\forall o \in ObjWInv \quad o.inv = true \Rightarrow o.Inv()$$

Implementation : InvCOP

- Reusable Monitor
- Aspect Generator
- At runtime:
  - Populate D by tracking p.f read during the execution of o.Inv()
  - if p.f changes, invoke Validate(p)

Default Binding

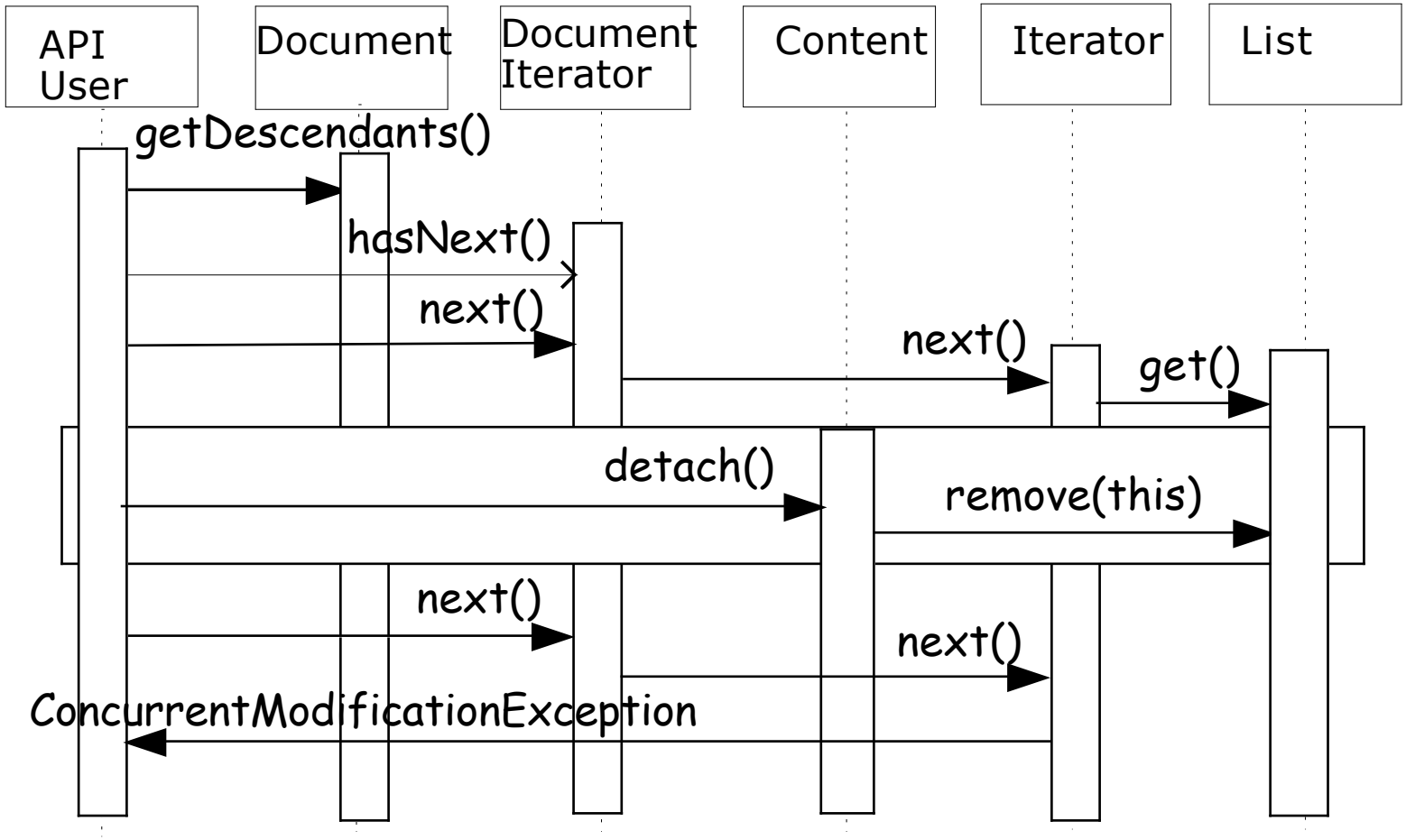
For objects o of type `ObjWInv`,

- invoke `Start(o)` after the construction of o
- invoke `Stop(o)` before every public method call on o
- invoke `Start(o)` after every public method call on o

Custom Binding Required

```
class T {
 public boolean Inv() {
    return 0 <= x && x < y;
 }

 public void method1() {
    x++;
    y++;
    user.m(this,..);
    ..
 }

public float method2() {
    return 1/(y-x);
 }
}


class User {
  public void m(T t,..) {
    //callback
    t.method2();
    ..
  }
}
```

Detecting Violations in JDOM



With InvCOP

```
java.lang.AssertionError: Invariant does not hold
   at rules.Inv_jdom.CheckAndSetInv(Inv_jdom.aj:122)
   ..
   at
   org.jdom.Element.removeContent(Element.java:885)
   at org.jdom.Content.detach(Content.java:91)
   at ItemHandler.processItem(OrderHandler.java:12)
   at
   OrderHandler.processOrder(OrderHandler.java:29)
```

Related Work

- JML Runtime Checker (Leavens et al)
  - Difference: JML checker does not report violations when an object's invariant is broken due to changes in dependents
- MOP checker (Rosu et al)
  - Difference: automated dependency tracking

Comparison with MOP

```
SafeEnum (Vector v, Enumeration+ e) {
[String location = "";]
event create<v,e>: end(call(Enumeration+.new(v,..))) with (e);
event updatesource<v>: end(call(* v.add*(..))) \/
                                end(call(* v.remove*(..))) \/ ...
                    {location = @LOC;}
event next<e>: begin(call(* e.nextElement()));
   formula : create next* updatesource+ next
}
validation handler { System.out.println("Vector updated at "
                                + @MONITOR.location); }
```

Ongoing Work

Extended InvCOP to enforce object protocols

- Various ownership protocols in the literature are expressed using a language ProLANG.
- Protocol Correctness is verified statically.
- Program Conformance is verified at runtime.