# Mining Operational Preconditions

Andrzej Wasylkowski • Andreas Zeller
Saarland University

**aspectj** *crosscutting objects for better modularity*

# bug.aj

```
@interface A {}

aspect Test {
  declare @field : @A int var* : @A;
  declare @field : int var* : @A;

  interface Subject {}

  public int Subject.vara;
  public int Subject.varb;
}

class X implements Test.Subject {}
```

# ajc Stack Trace

```
java.util.NoSuchElementException
at java.util.AbstractList$Itr.next(AbstractList.java:427)
at org.aspectj.weaver.bcel.BcelClassWeaver.
  weaveAtFieldRepeatedly(BcelClassWeaver.java:1016)
…
```

# ajc Stack Trace

```
java.util.NoSuchElementException
at java.util.AbstractList$Itr.next(AbstractList.java:427)
at org.aspectj.weaver.bcel.BcelClassWeaver.
  weaveAtFieldRepeatedly(BcelClassWeaver.java:1016)
…
```

# weaveAtFieldRepeatedly

```
for (Iterator it = c1.iterator(); it.hasNext();) {
  E e1 = (E) it.next();
  …
  for (Iterator it2 = c2.iterator(); it.hasNext();) {
    E e2 = (E) it2.next();
    …
  }
}
```

# weaveAtFieldRepeatedly

```
for (Iterator it = c1.iterator(); it.hasNext();) {
  E e1 = (E) it.next();
  …
  for (Iterator it2 = c2.iterator(); it.hasNext();) {
    E e2 = (E) it2.next();
    …
  }
}
```

*should be* `it2`

# weaveAtFieldRepeatedly

```
for (Iterator it = c1.iterator(); it.hasNext();) {
  E e1 = (E) it.next();
  …
  for (Iterator it2 = c2.iterator(); it.hasNext();) {
    E e2 = (E) it2.next();
    …
  }
}
```

*should be* `it2`

- Invalid iterator usage:
  hasNext() should precede next()

# Preconditions

- Invoking next() with no next element violates a *precondition*
- Traditional preconditions are axiomatic – describing the state of the system
- How do we reach that state?

# Preconditions

```
close (int fildes)
```

# Preconditions

`close (int fildes)`

- Axiomatic: *fildes* is a valid file descriptor

# Preconditions

`close (int fildes)`

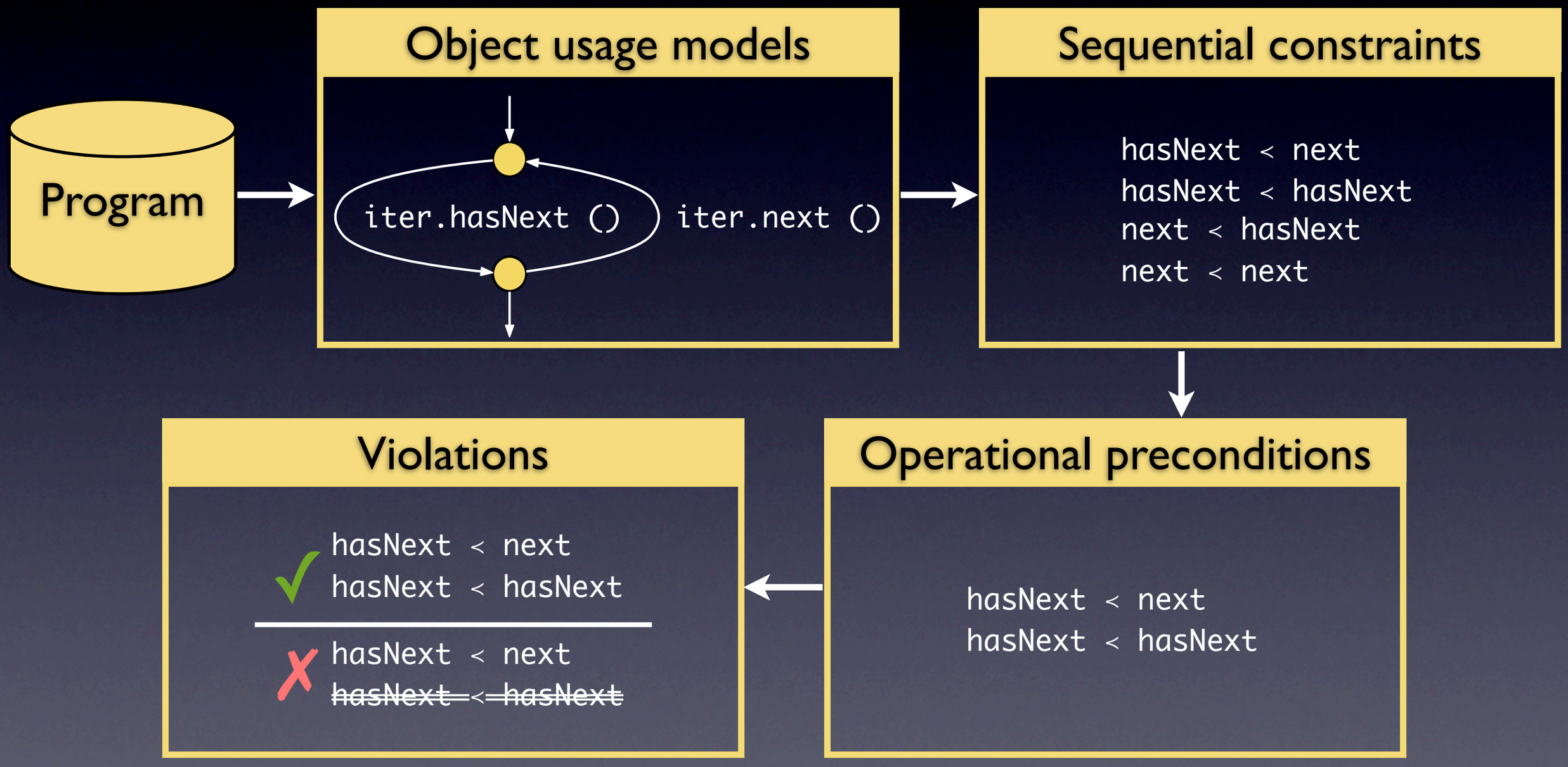
- Axiomatic: *fildes* is a valid file descriptor
- Operational: *fildes* stems from a call to open() with read() and write() calls in between

# Preconditions

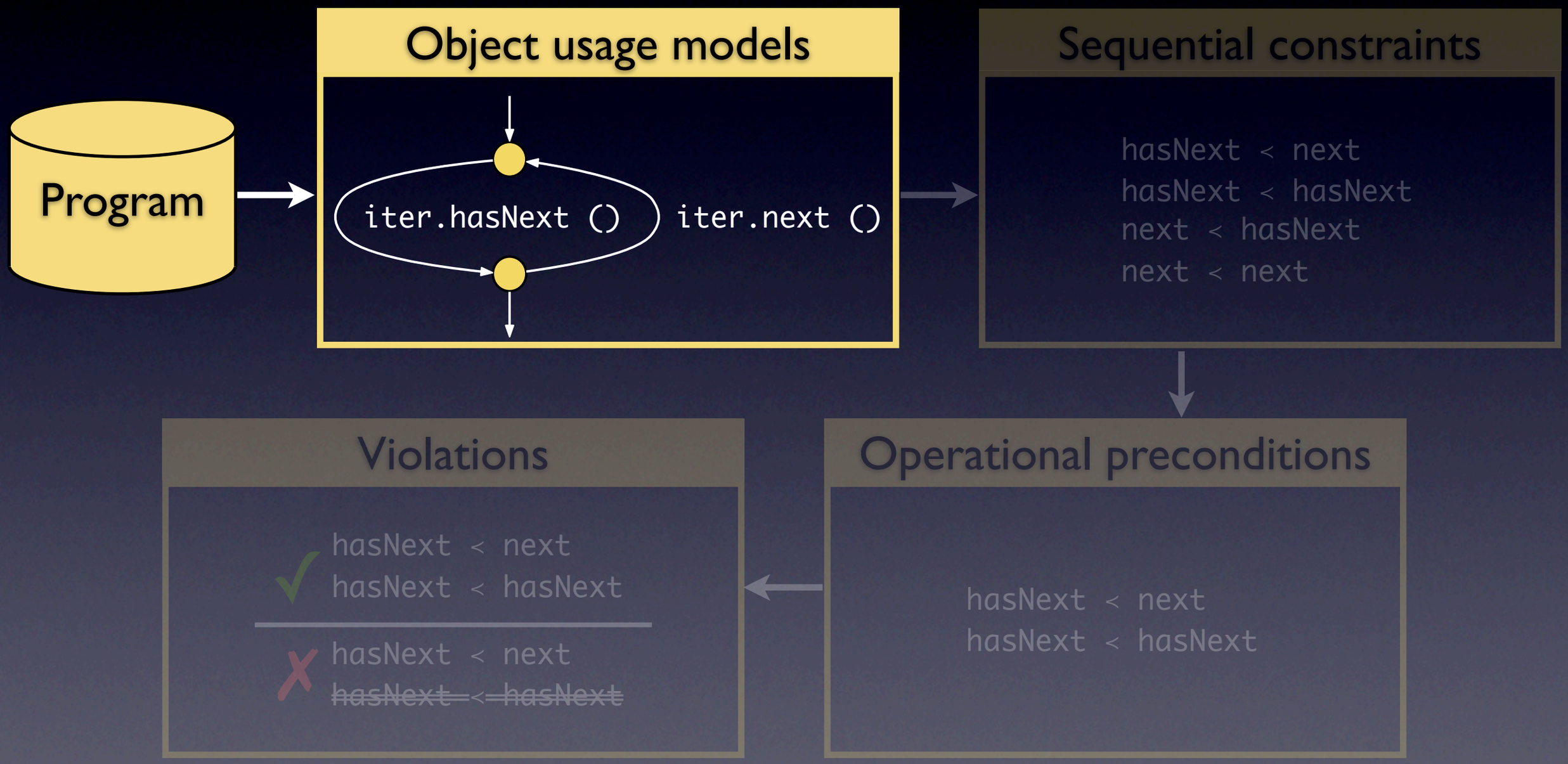
`close (int fildes)`

- Axiomatic: *fildes* is a valid file descriptor
- Operational: *fildes* stems from a call to open() with read() and write() calls in between
- Can we learn and check operational preconditions?

# OP-Miner


Program
Object usage models
iter.hasNext ()
iter.next ()
Sequential constraints
hasNext ≺ next
hasNext ≺ hasNext
next ≺ hasNext
next ≺ next
Operational preconditions
hasNext ≺ next
hasNext ≺ hasNext
Violations
hasNext ≺ next
hasNext ≺ hasNext
hasNext ≺ next
hasNext ≺ hasNext

# OP-Miner

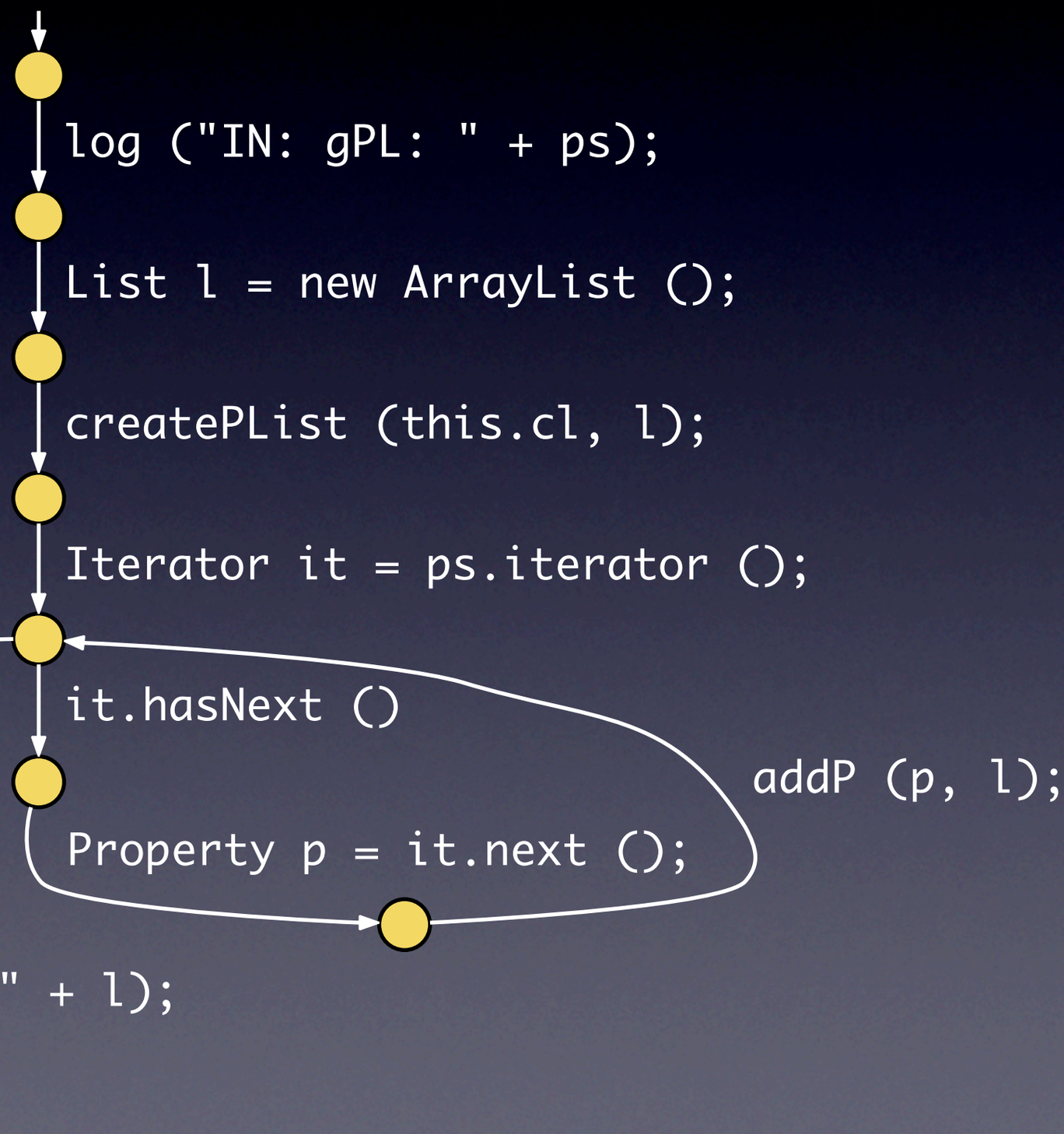

Program
Object usage models
iter.hasNext ()
iter.next ()
Sequential constraints
hasNext < next
hasNext < hasNext
next < hasNext
next < next
Operational preconditions
hasNext < next
hasNext < hasNext
Violations
hasNext < next
hasNext < hasNext
hasNext < next
hasNext < hasNext

# Creating a method model

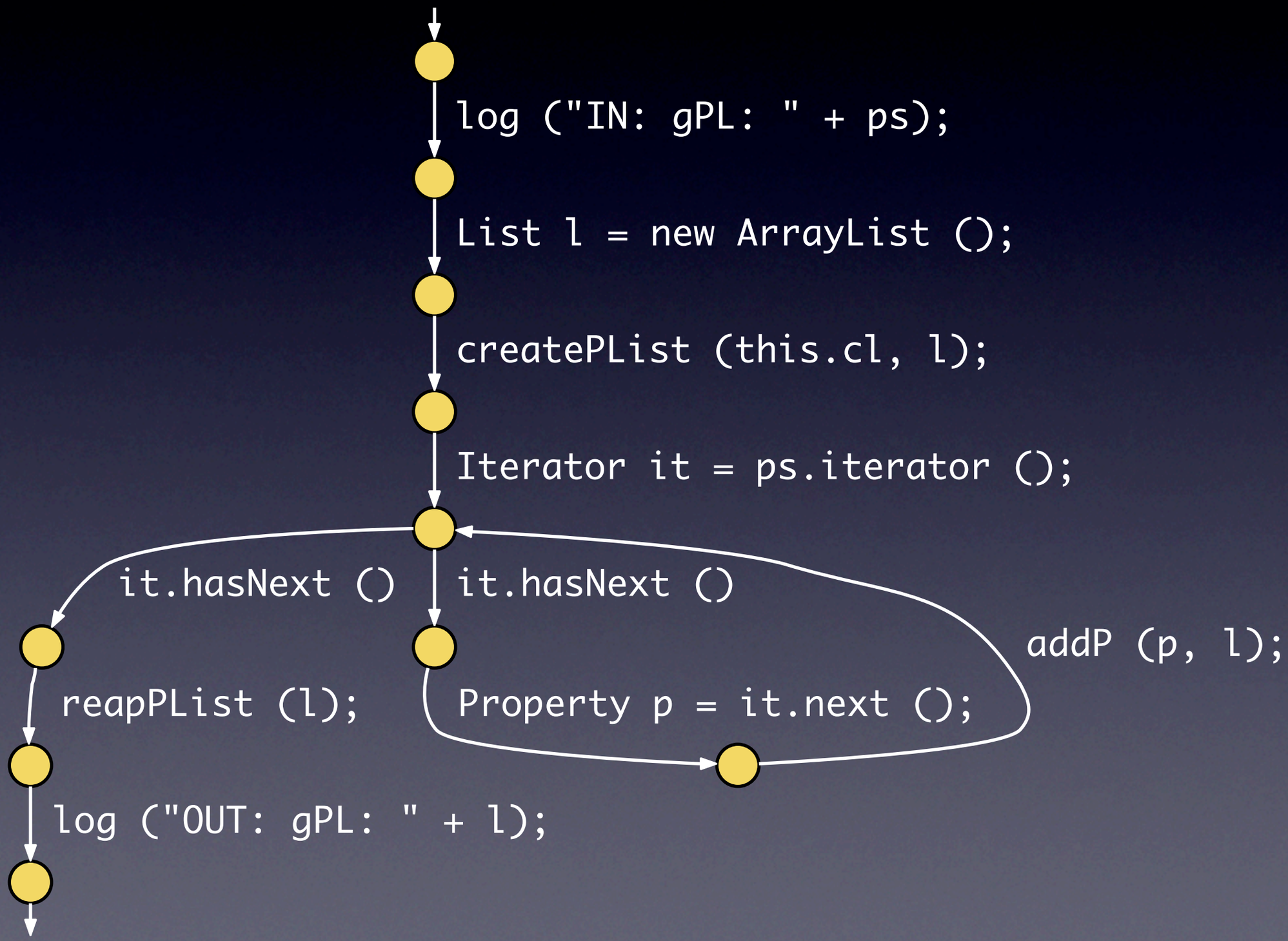
```
public List gPL (Set ps) {
  log (“IN: gPL: “ + ps);
  List l = new ArrayList ();
  createPList (this.cl, l);
  Iterator it = ps.iterator ();
  while (it.hasNext ()) {
    Property p = it.next ();
    addP (p, l);
  }
  reapPList (l);
  log (“OUT: gPL: “ + l);
  return l;
}
```

# Creating a method model

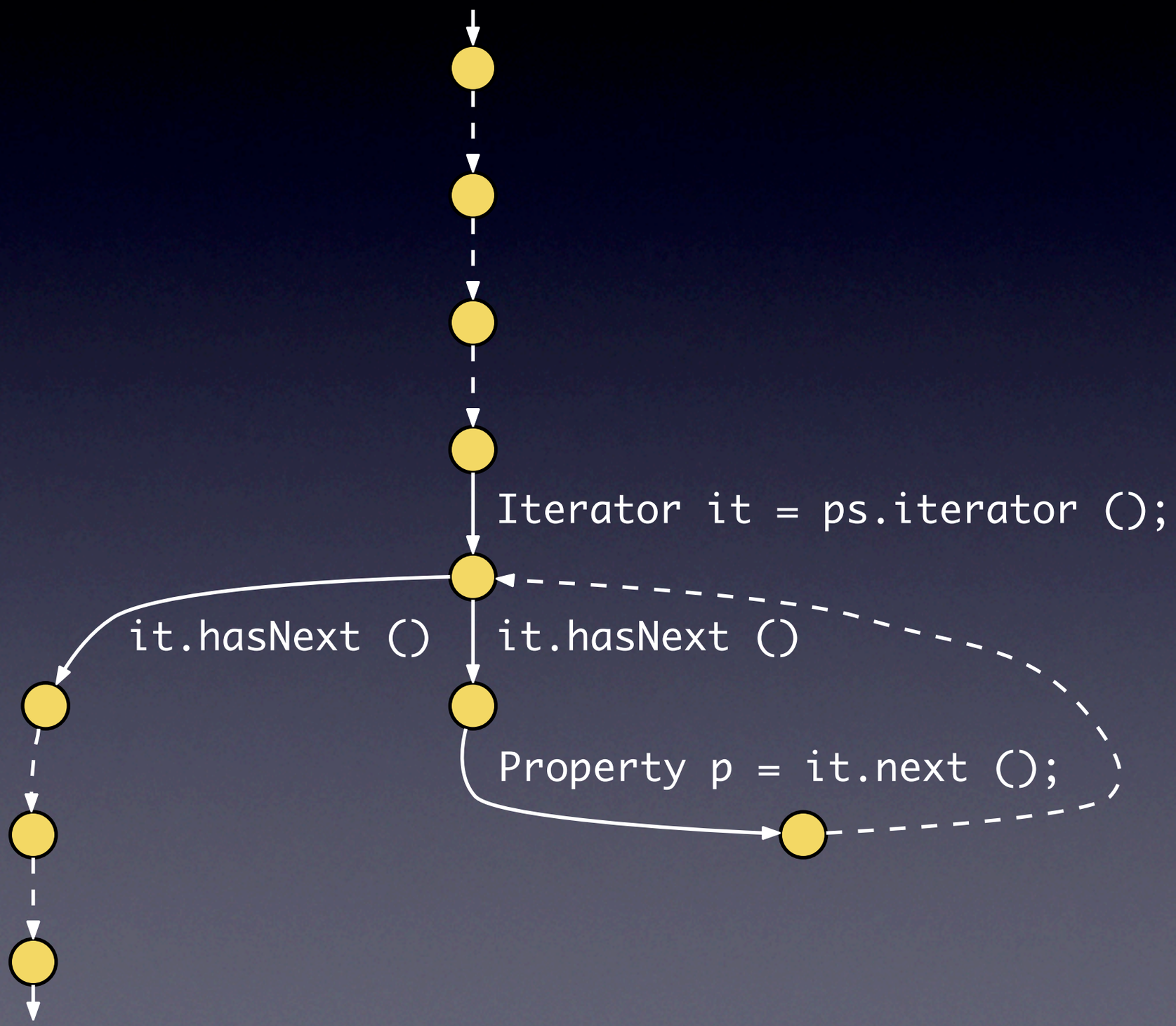
```
public List gPL (Set ps) {
  log (“IN: gPL: “ + ps);
  List l = new ArrayList ();
  createPList (this.cl, l);
  Iterator it = ps.iterator ();
  while (it.hasNext ()) {
    Property p = it.next ();
    addP (p, l);
  }
  reapPList (l);
  log (“OUT: gPL: “ + l);
  return l;
}
```



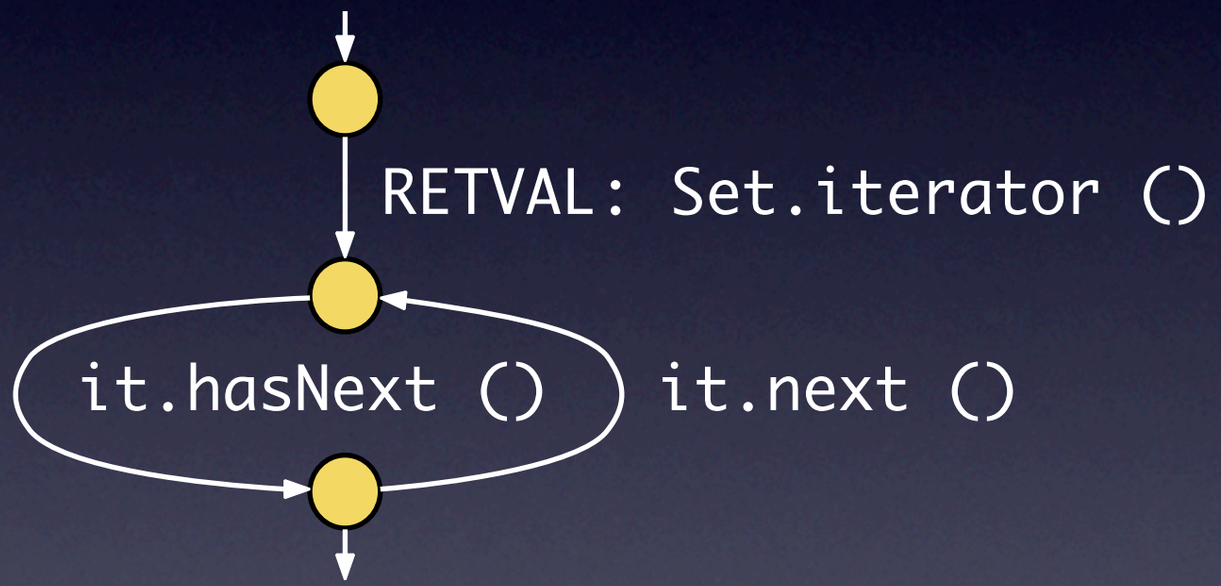
# Creating a usage model



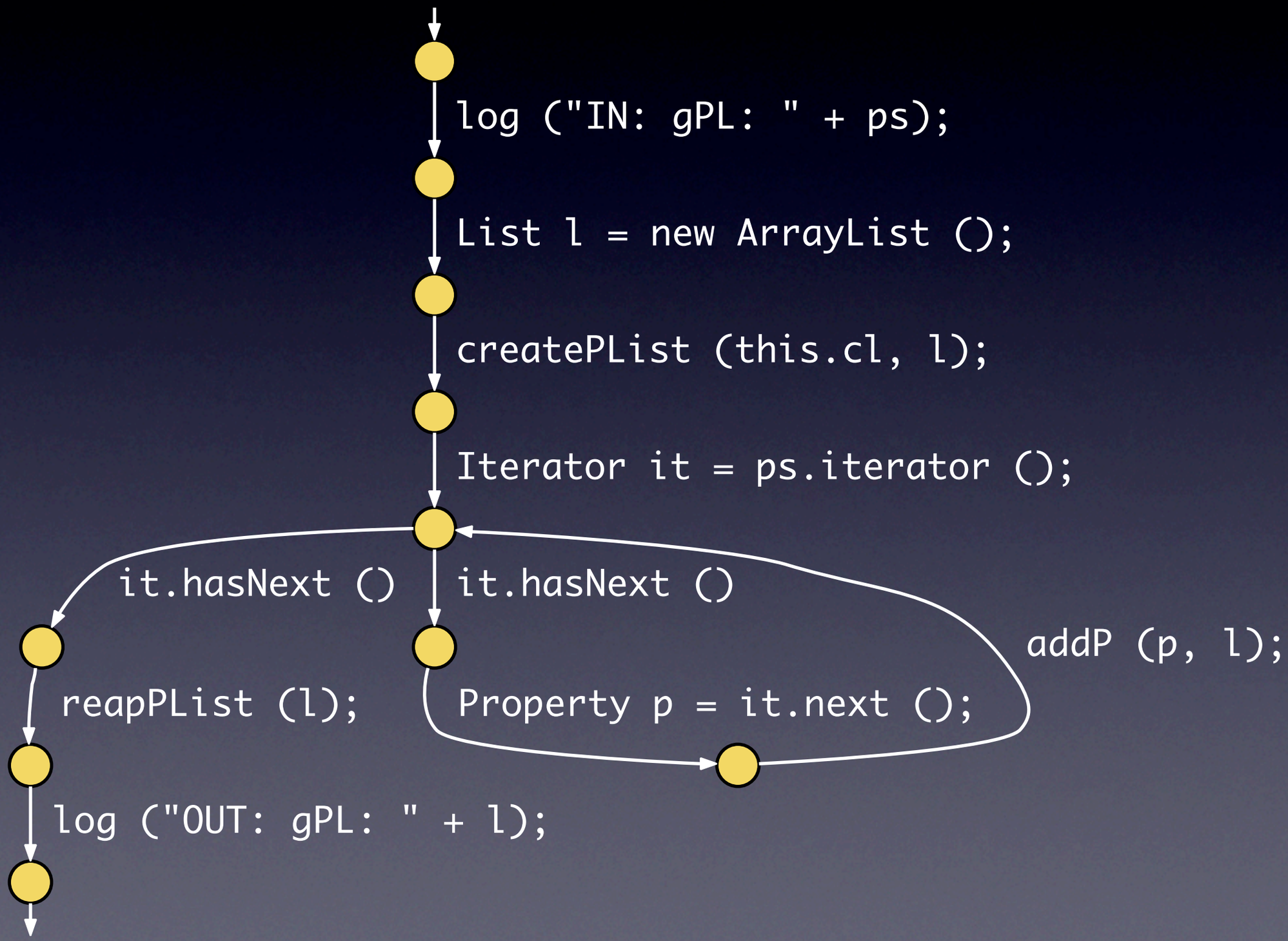
# Creating a usage model



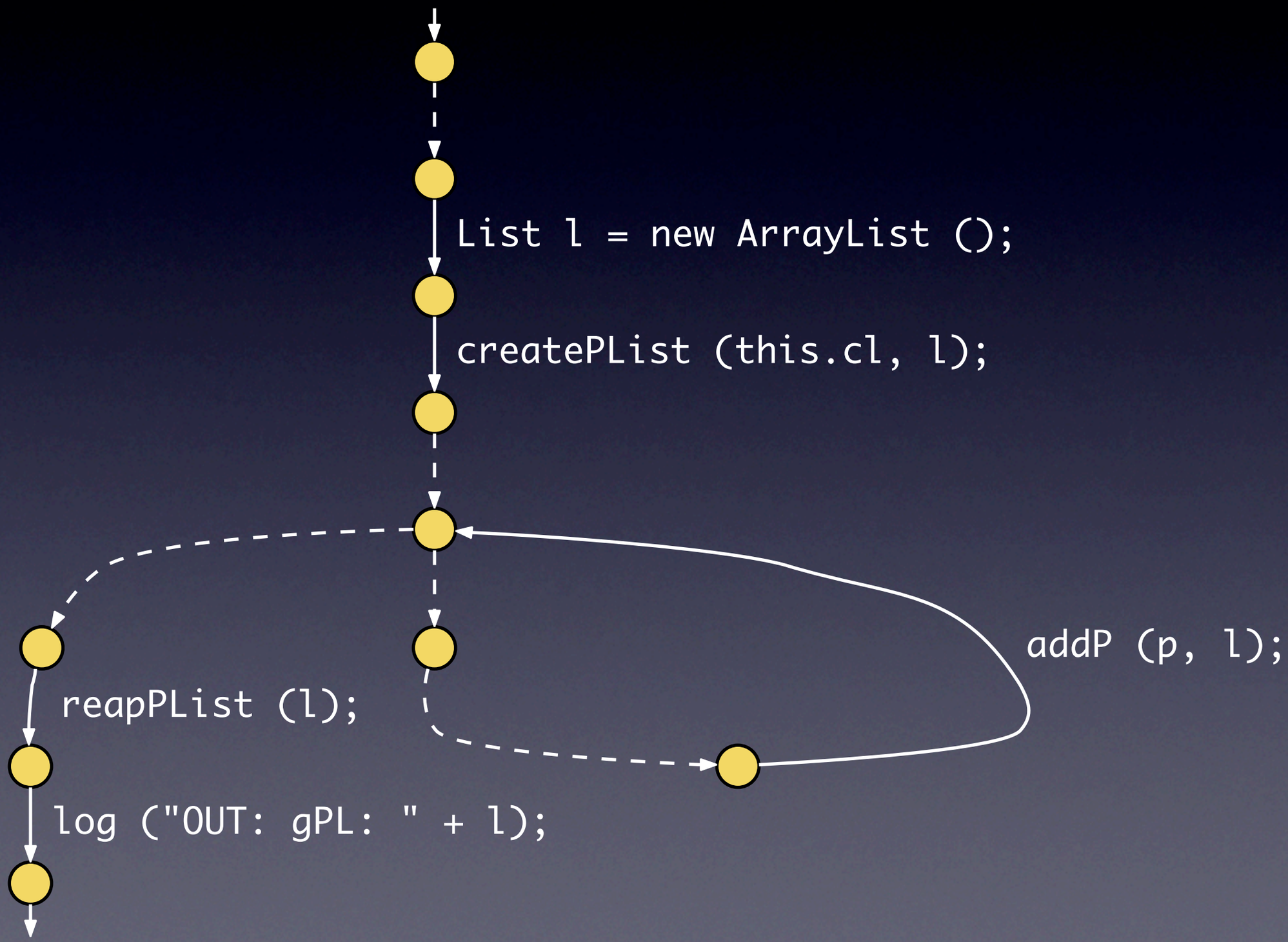
# Creating a usage model



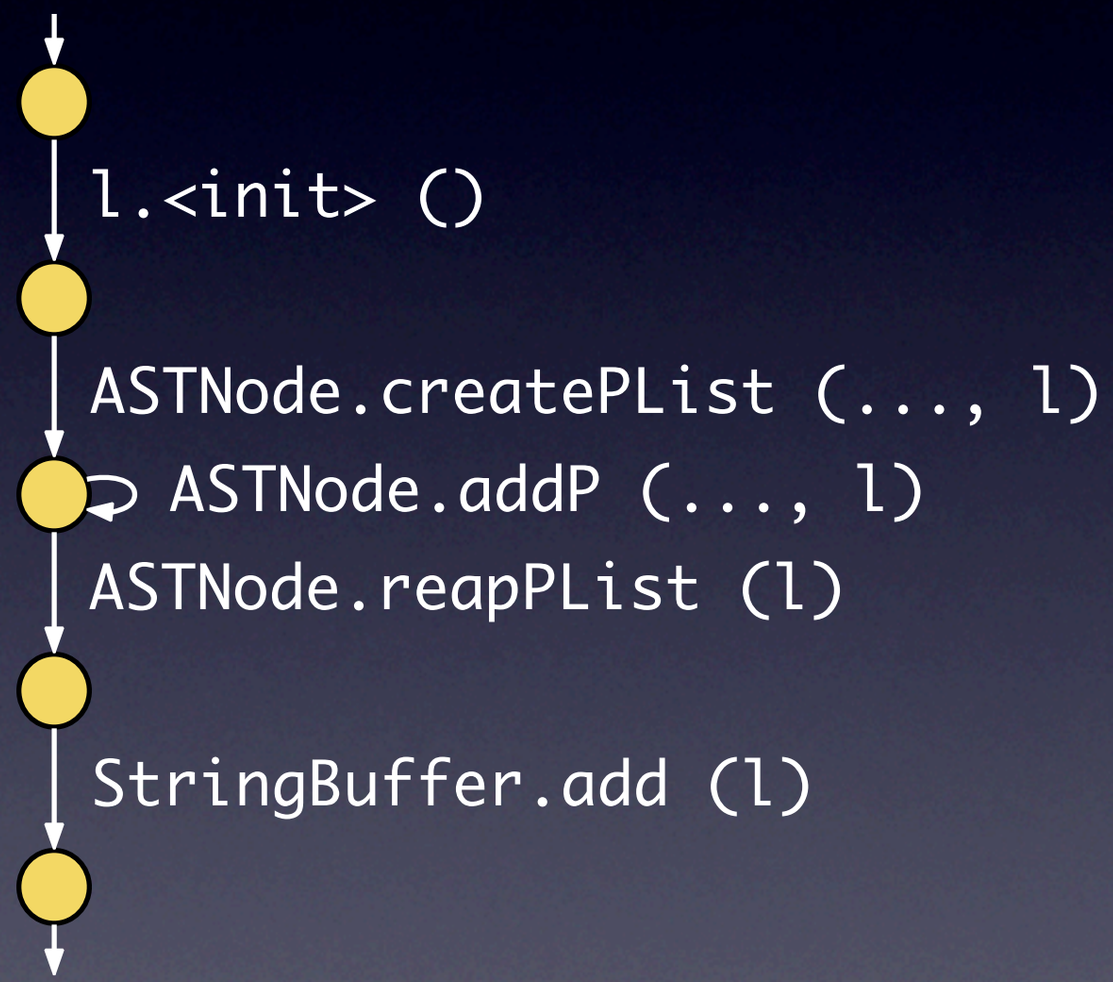
# Creating a usage model



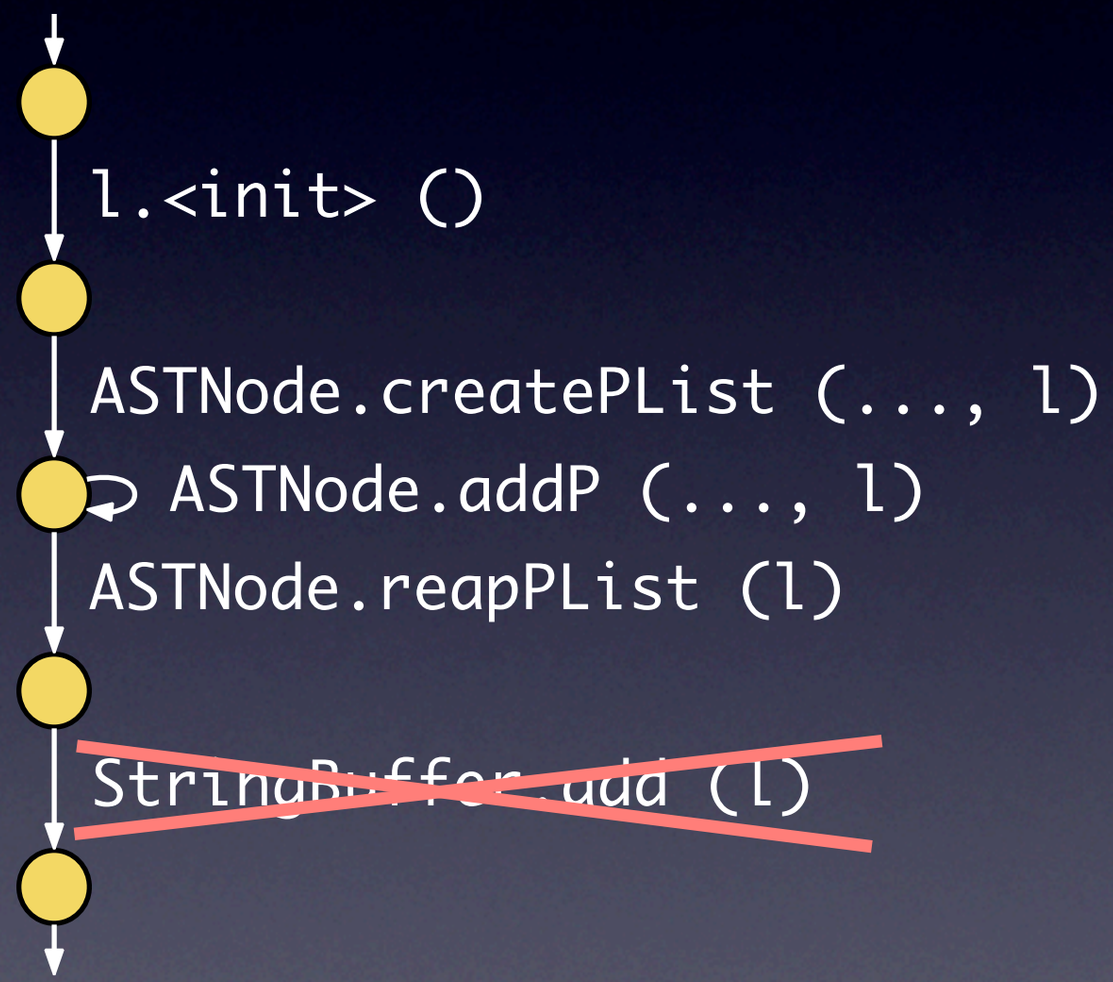
# Creating a usage model



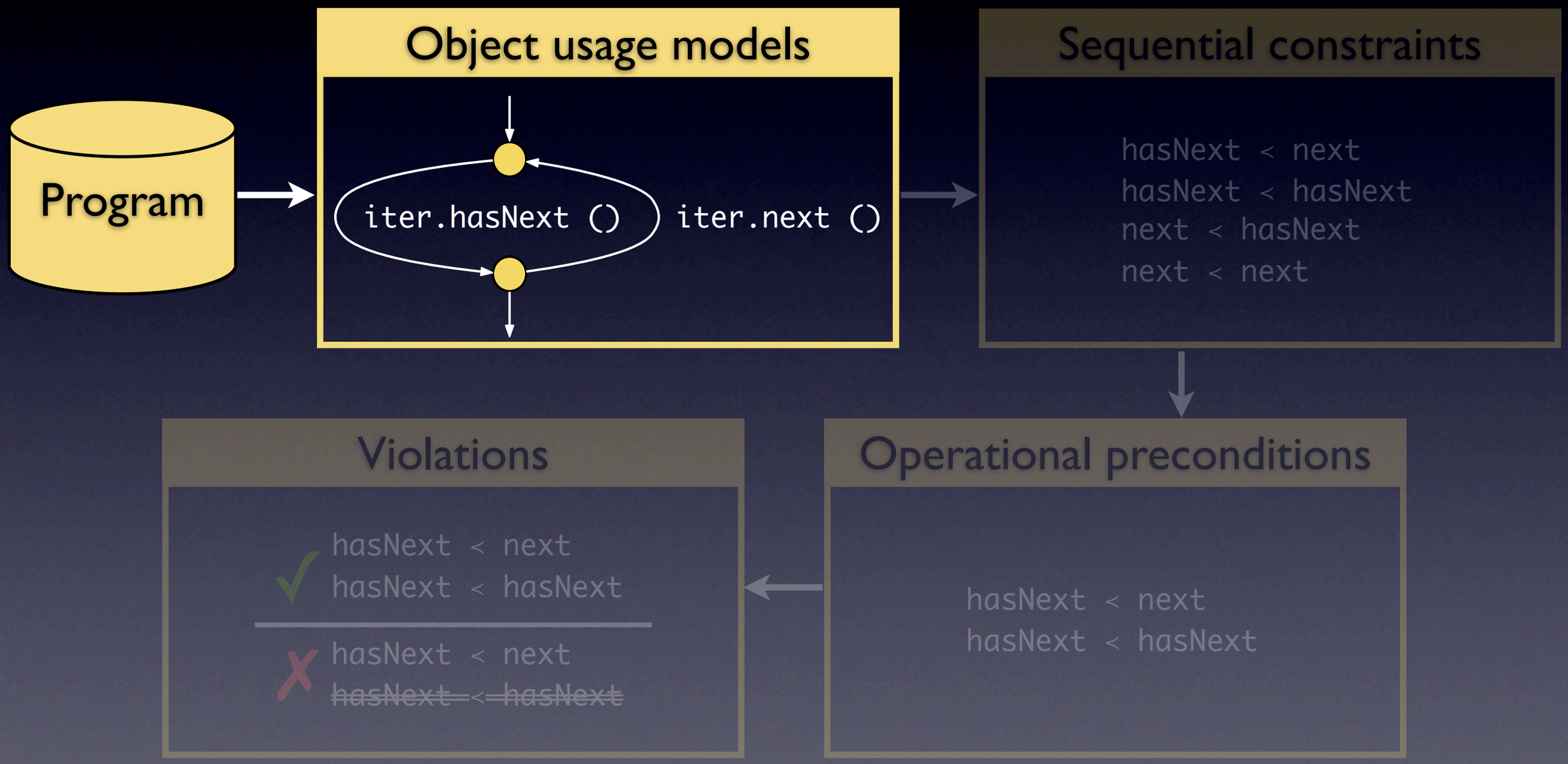
# Creating a usage model



# Creating a usage model



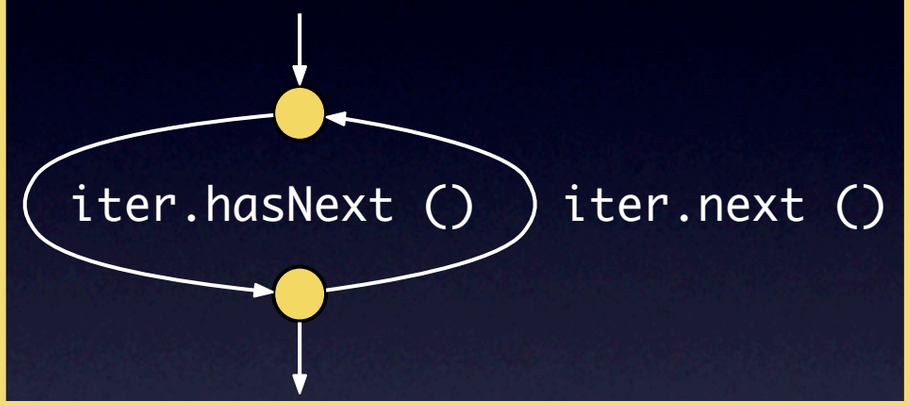
# OP-Miner


Program
Object usage models
iter.hasNext ()
iter.next ()
Sequential constraints
hasNext < next
hasNext < hasNext
next < hasNext
next < next
Operational preconditions
hasNext < next
hasNext < hasNext
Violations
hasNext < next
hasNext < hasNext
hasNext < next
hasNext < hasNext

# OP-Miner

Program

## Object usage models




## Sequential constraints

```
hasNext ≺ next
hasNext ≺ hasNext
next ≺ hasNext
next ≺ next
```

## Operational preconditions

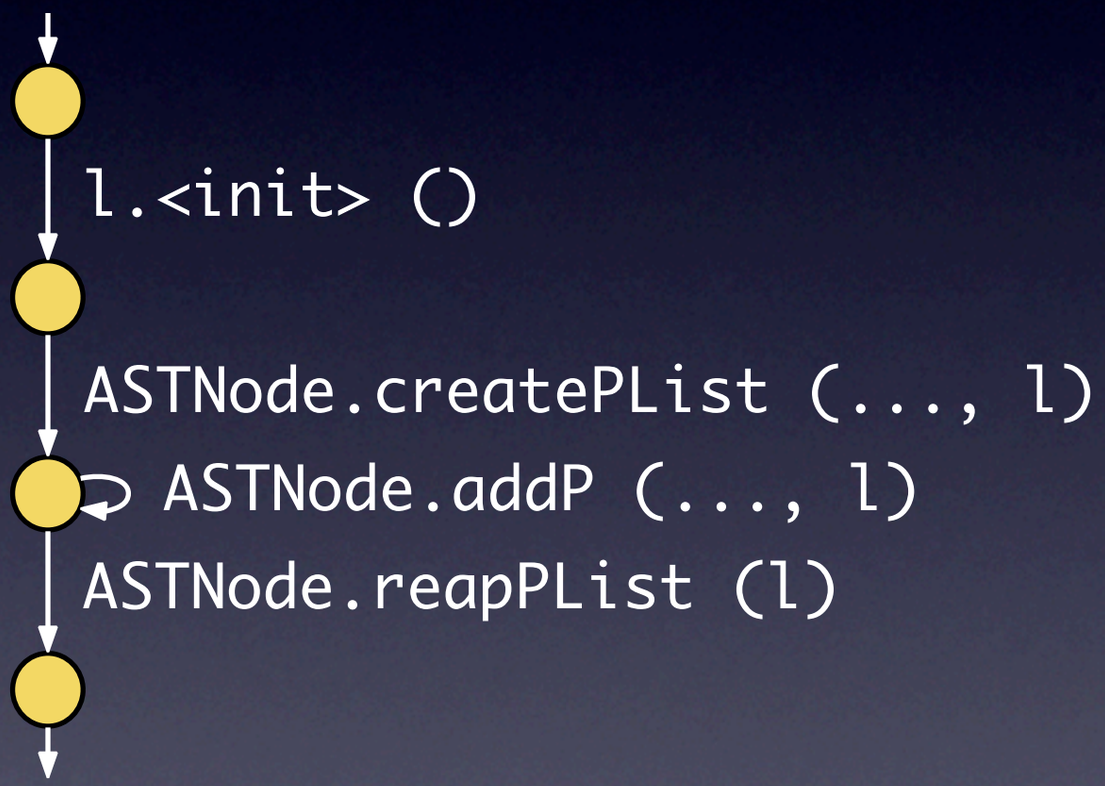
```
hasNext ≺ next
hasNext ≺ hasNext
```

## Violations

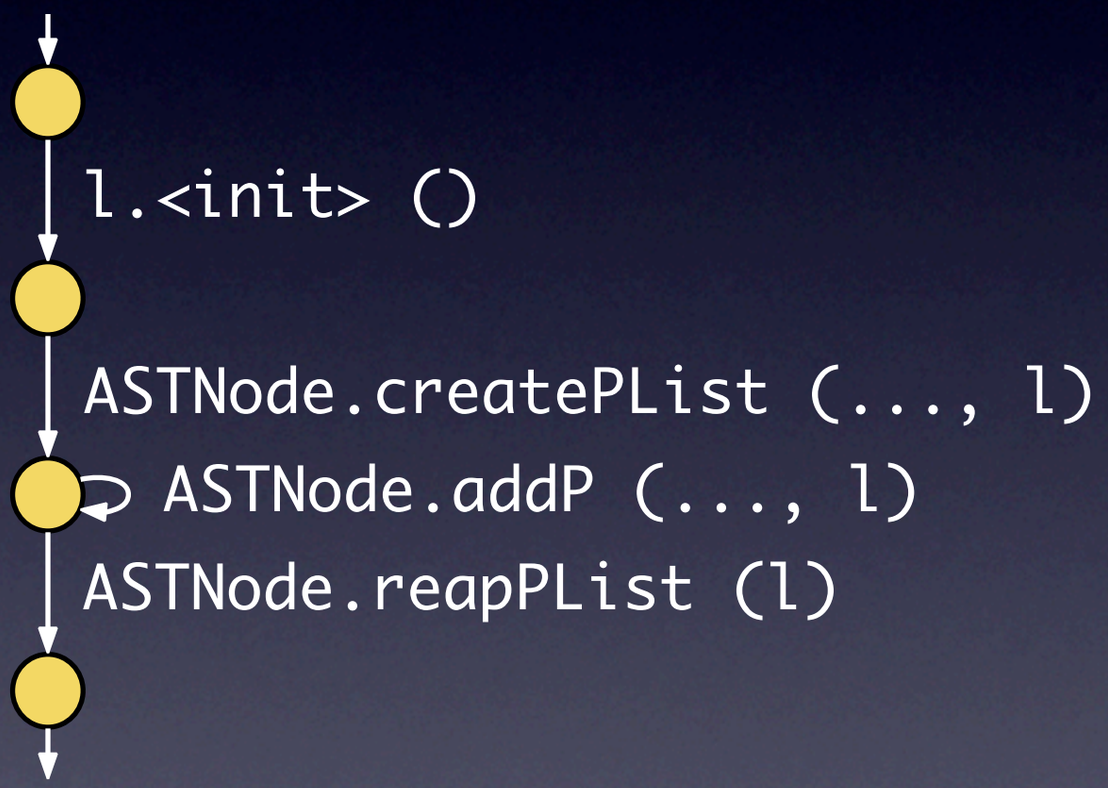
✓
```
hasNext ≺ next
hasNext ≺ hasNext
```

✗
```
hasNext ≺ next
~~hasNext ≺ hasNext~~
```

# Extracting sequential constraints

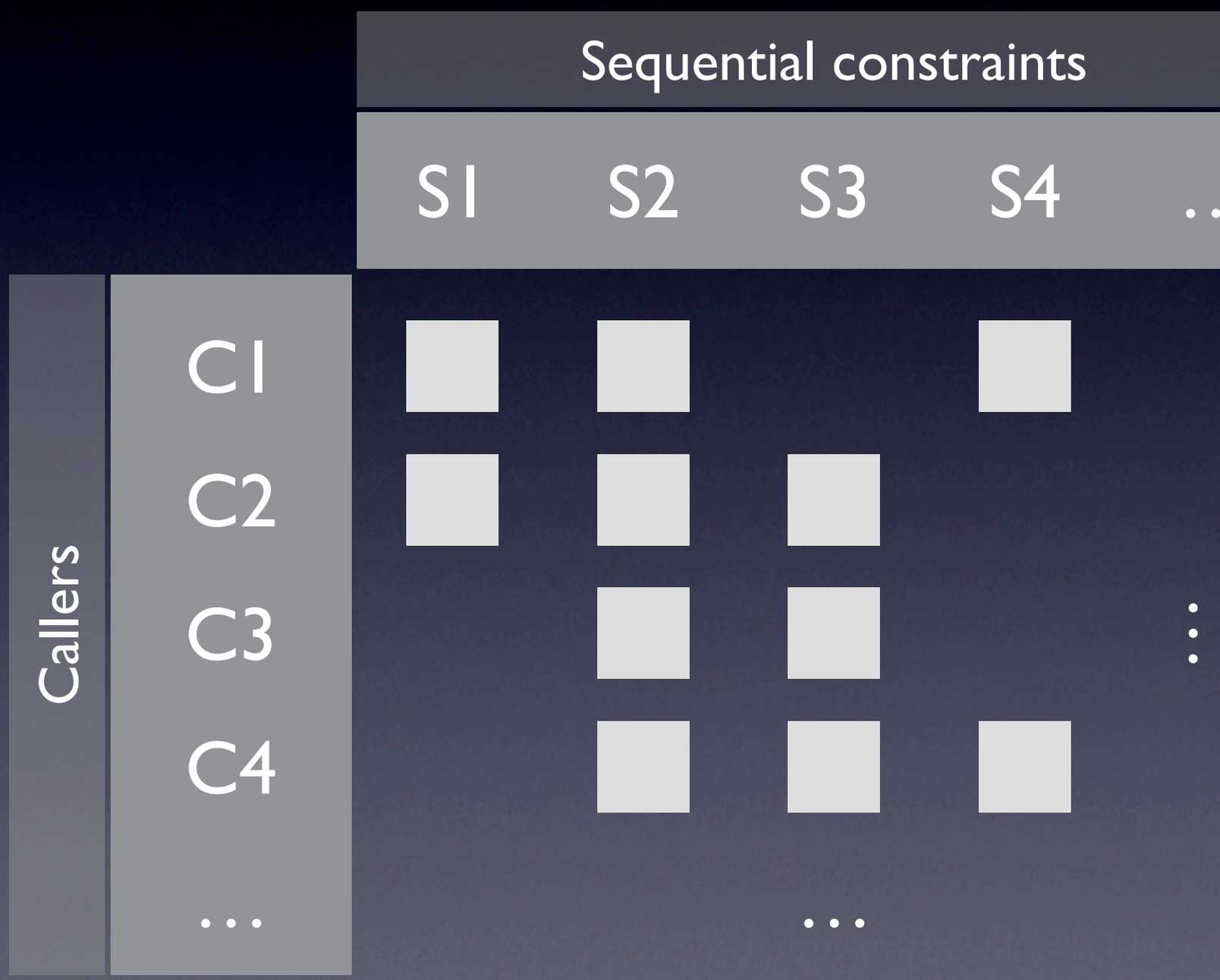


# Extracting sequential constraints

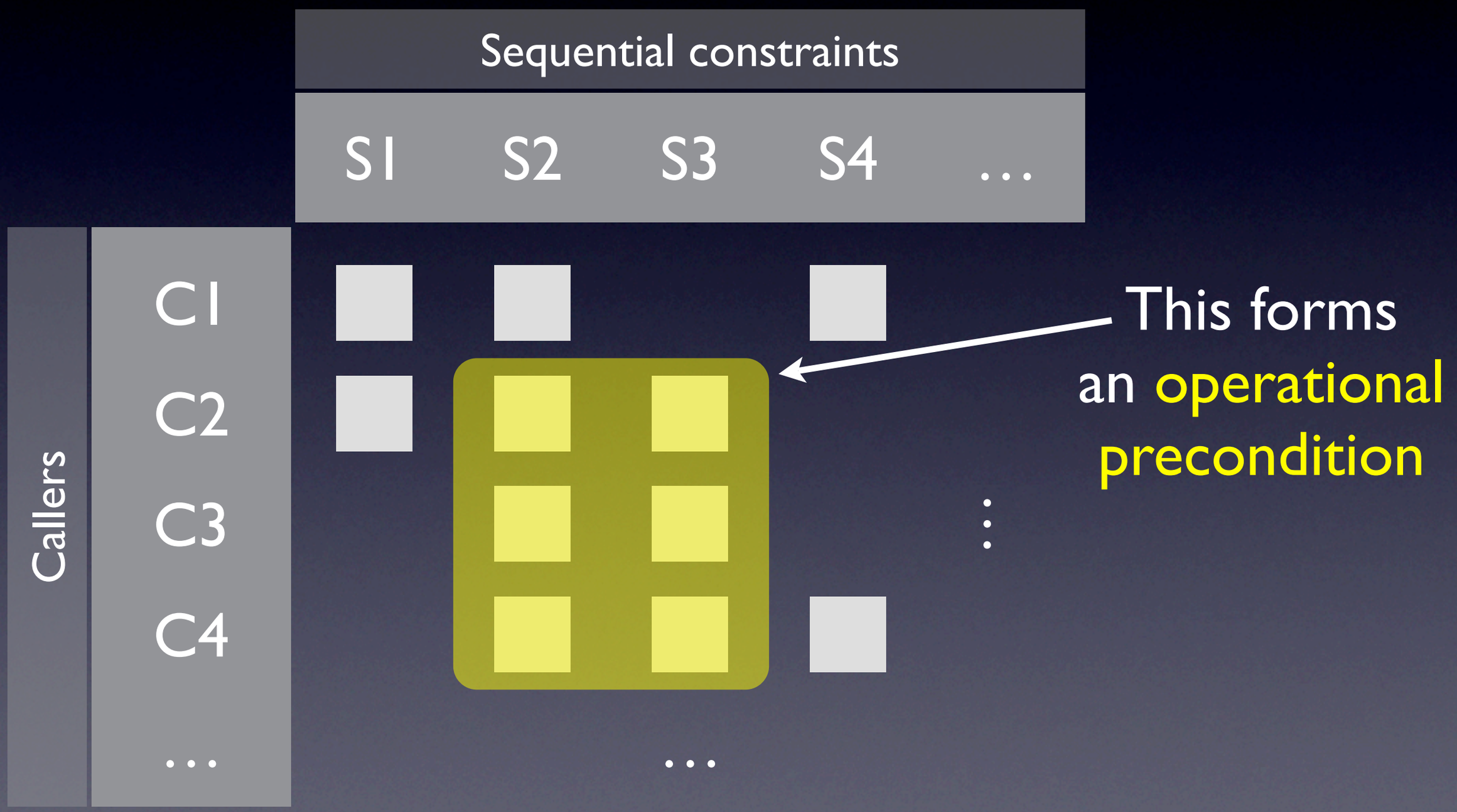



```
l.<init> () ≺ createPList (…, l)
l.<init> () ≺ addP (…, l)
l.<init> () ≺ reapPList (l)
createPList (…, l) ≺ addP (…, l)
createPList (…, l) ≺ reapPList (l)
addP (…, l) ≺ addP (…, l)
addP (…, l) ≺ reapPList (l)
```

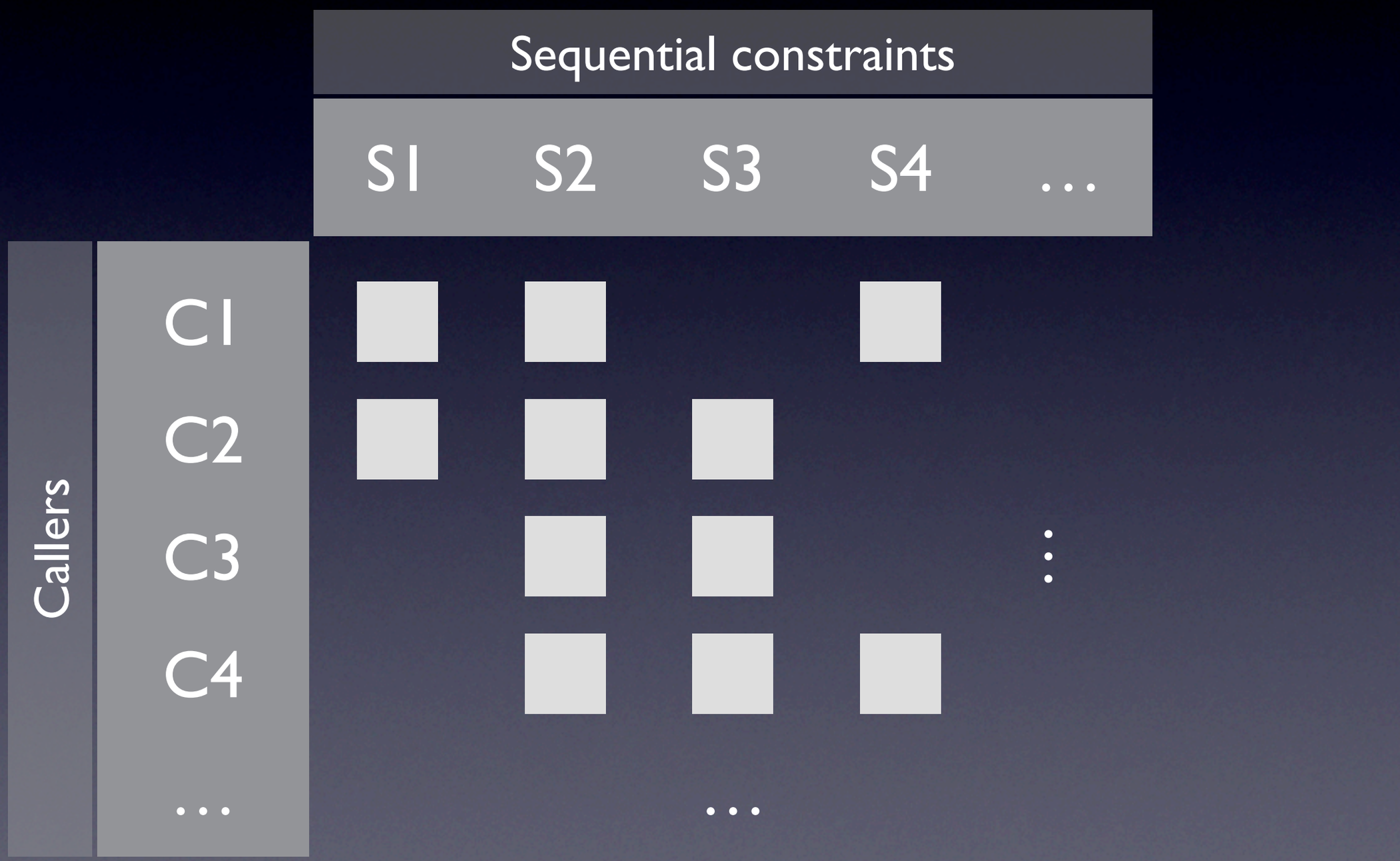
# Callers vs. constraints



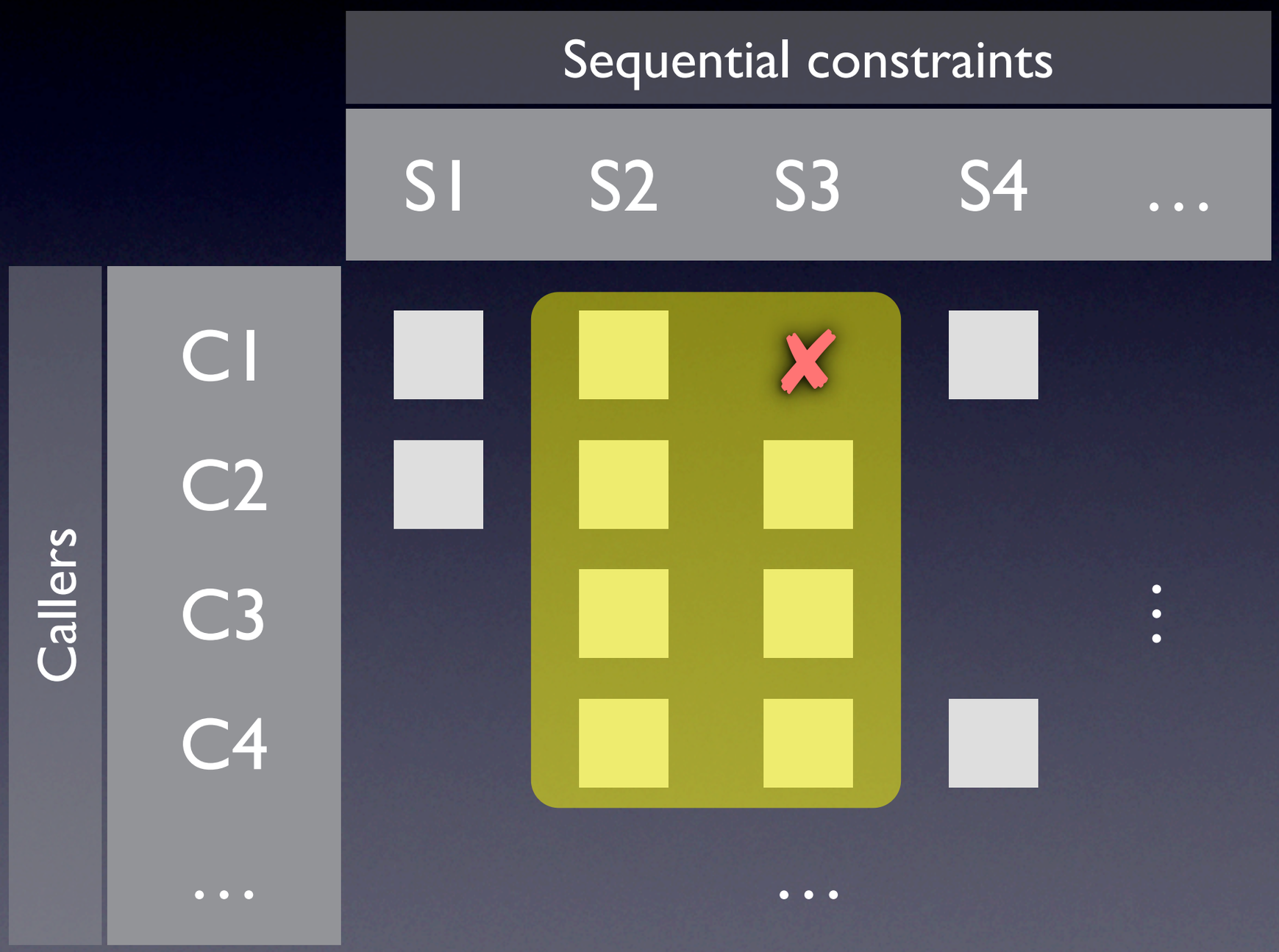
# Callers vs. constraints



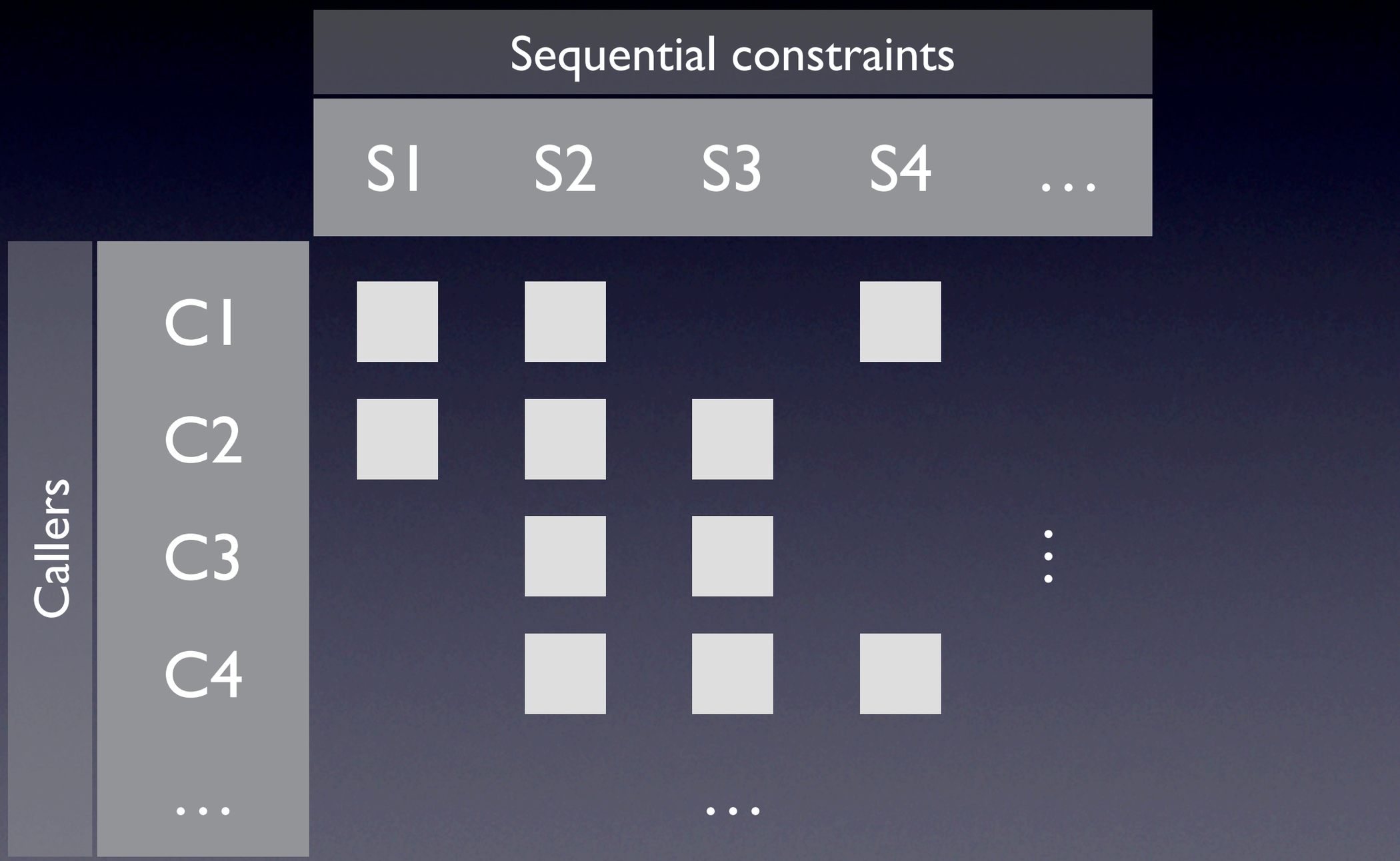
# Detecting violations



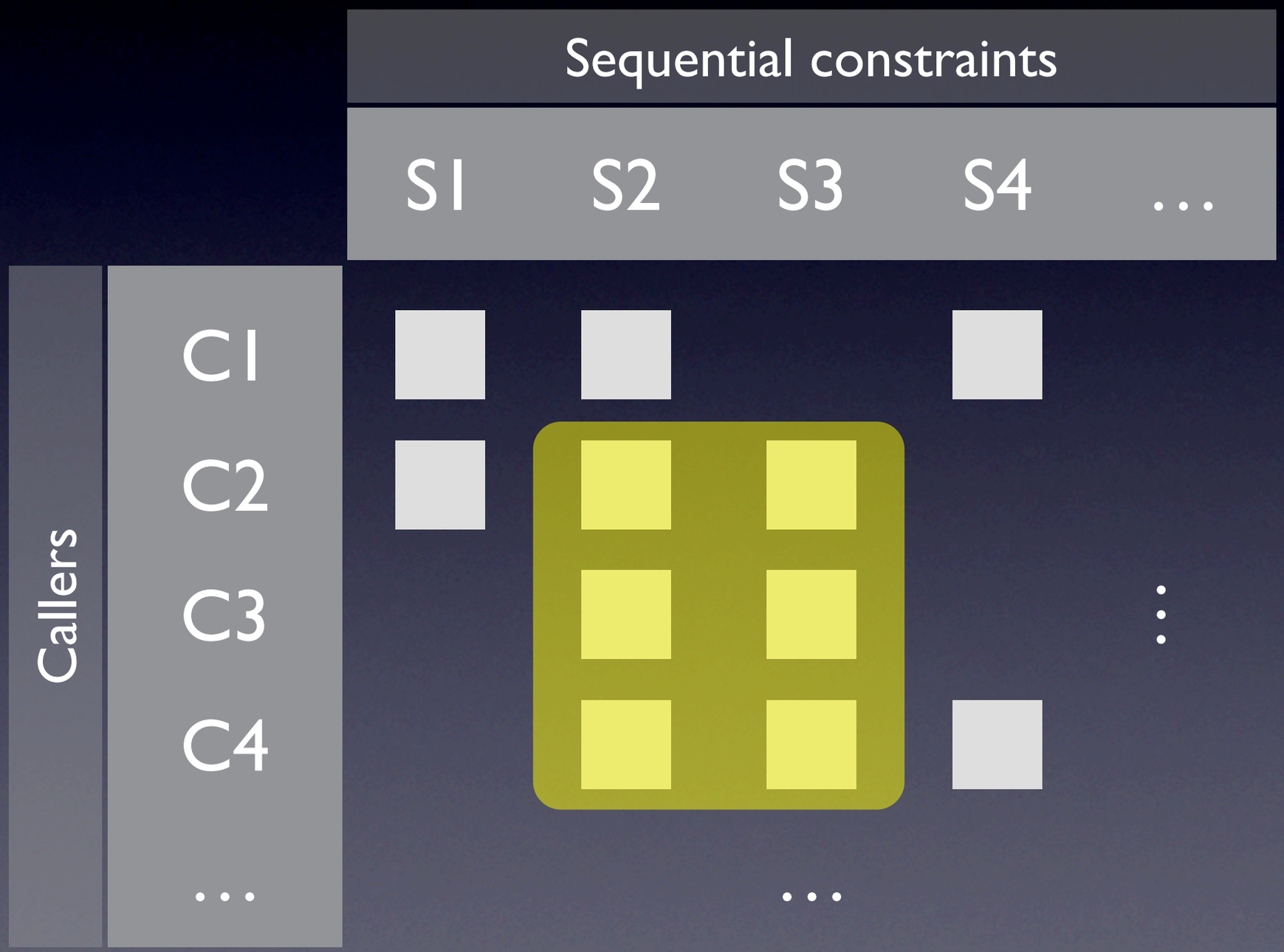
# Detecting violations



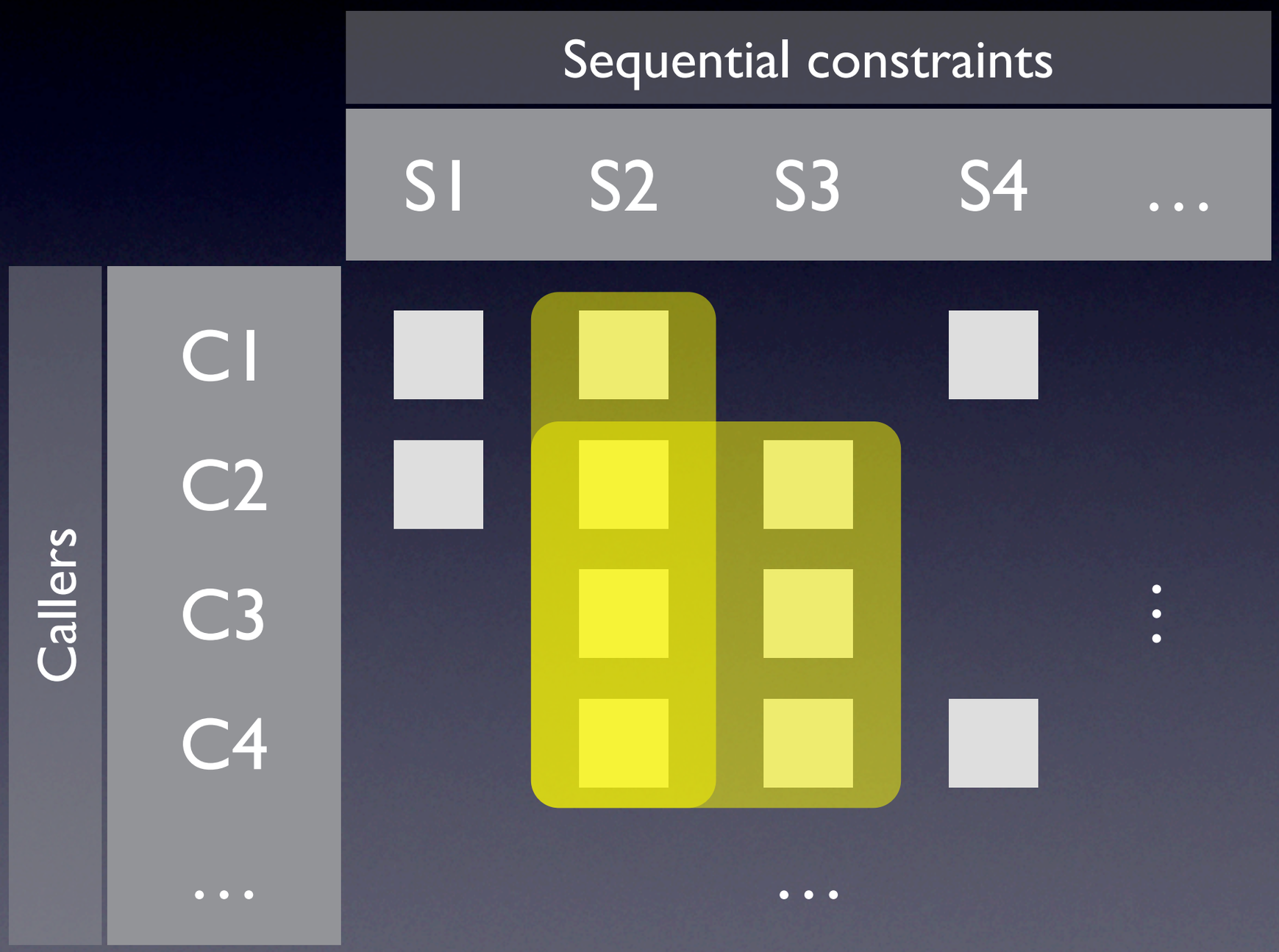
# Detecting violations



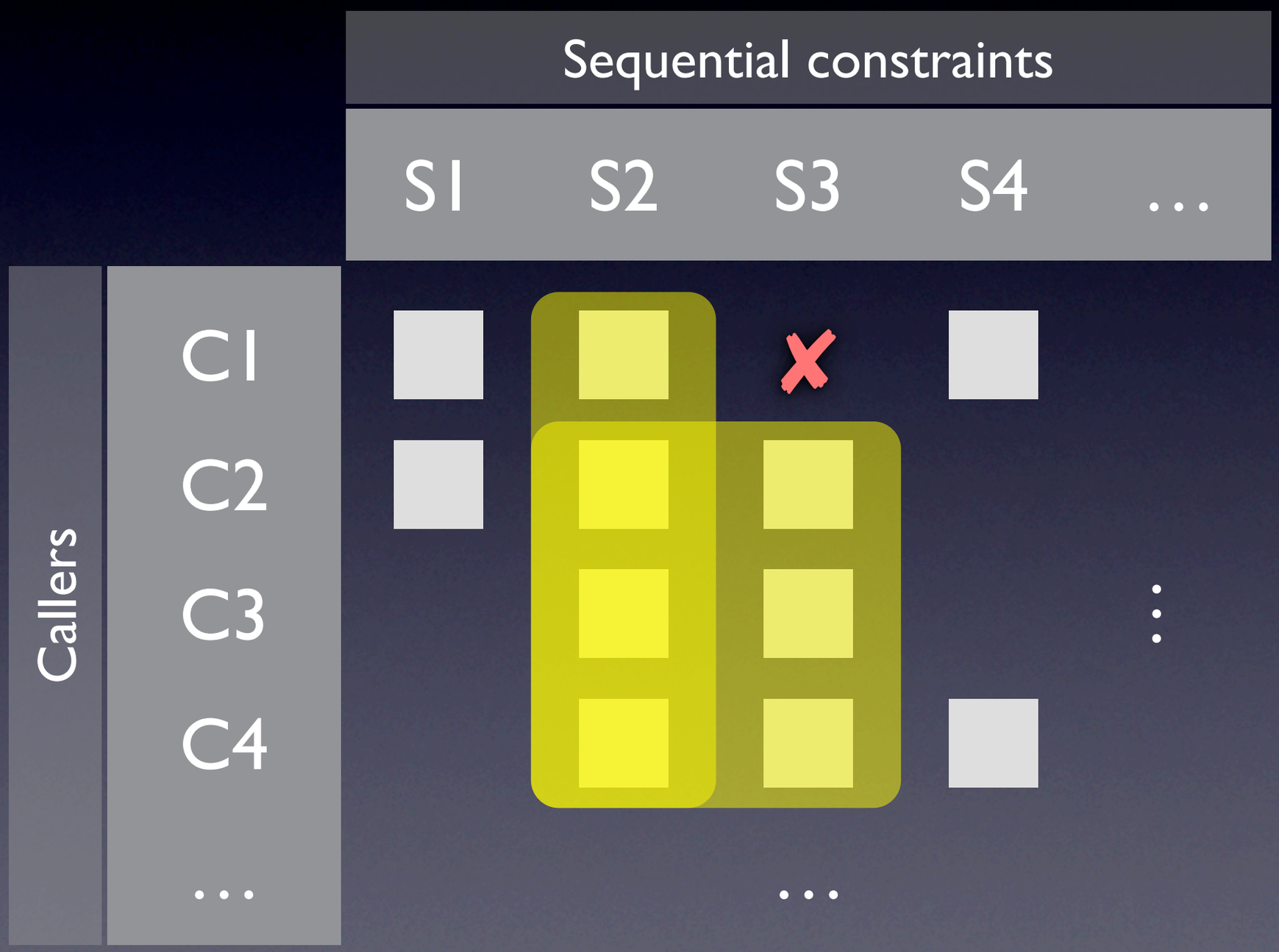
# Detecting violations



# Detecting violations



# Detecting violations



# Case study: AspectJ

- Compiler for the AspectJ language
- 36,045 methods in 2,957 classes
- 1,154 methods with OP support ≥ 20
- 300 violations found in 8 minutes

# A defect

```
for (Iterator it = c1.iterator(); it.hasNext();) {
  E e1 = (E) it.next();
  …
  for (Iterator it2 = c2.iterator(); it.hasNext();) {
    E e2 = (E) it2.next();
    …
  }
}
```

# A defect

```
for (Iterator it = c1.iterator(); it.hasNext();) {
  E e1 = (E) it.next();
  …
  for (Iterator it2 = c2.iterator(); it.hasNext();) {
    E e2 = (E) it2.next();
    …
  }
}
```

*should be* `it2`

# Another defect

```
public void visitNEWARRAY (NEWARRAY o) {
  byte t = o.getTypecode ();
  if (! ( (t == Constants.T_BOOLEAN) ||
          (t == Constants.T_CHAR) ||
          …
          (t == Constants.T_LONG) ) ) {
    constraintViolated (o, “(…) ‘+t+’ (…)”);
  }
}
```

# Another defect

```
public void visitNEWARRAY (NEWARRAY o) {
  byte t = o.getTypecode ();
  if (! ( (t == Constants.T_BOOLEAN) ||
          (t == Constants.T_CHAR) ||
          …
          (t == Constants.T_LONG) ) ) {
    constraintViolated (o, “(…) ‘+t+’ (…)”);
  }
}
```

*should be using double quotes*

# A false positive

```
Name internalNewName (String[] identifiers) {
  …
  for (int i = 1; i < count; i++) {
    SimpleName name = new SimpleName (this);
    name.internalSetIdentifier (identifiers[i]);
    …
  }
  …
}
```

# A false positive

```
Name internalNewName (String[] identifiers) {
  …
  for (int i = 1; i < count; i++) {
    SimpleName name = new SimpleName (this);
    name.internalSetIdentifier (identifiers[i]);
    …
  }
  …
}
```

*should stay as is*

# A code smell

```
public String getRetentionPolicy () {
  …
  for (Iterator it = …; it.hasNext();) {
    … = it.next ();
    …
    return retentionPolicy;
  }
  …
}
```

# A code smell

```
public String getRetentionPolicy () {
  …
  for (Iterator it = …; it.hasNext();) {
    … = it.next ();
    …
    return retentionPolicy;
  }
  …
}
```

*should be fixed*

# AspectJ violations

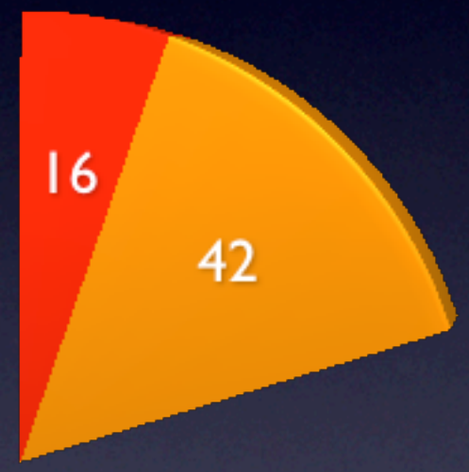
Defects  Code smells  False positives

# AspectJ violations

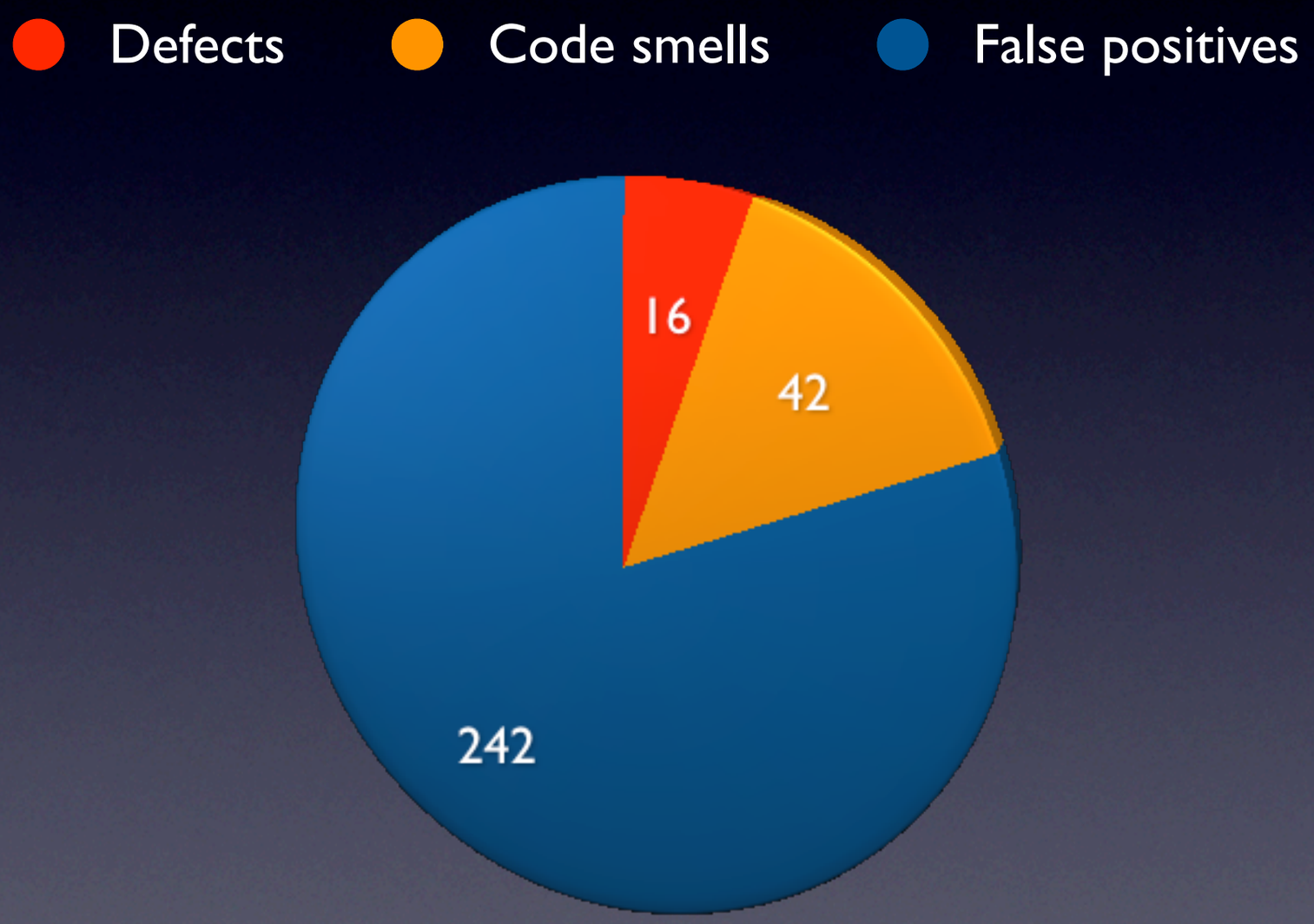
Defects  Code smells  False positives

# AspectJ violations

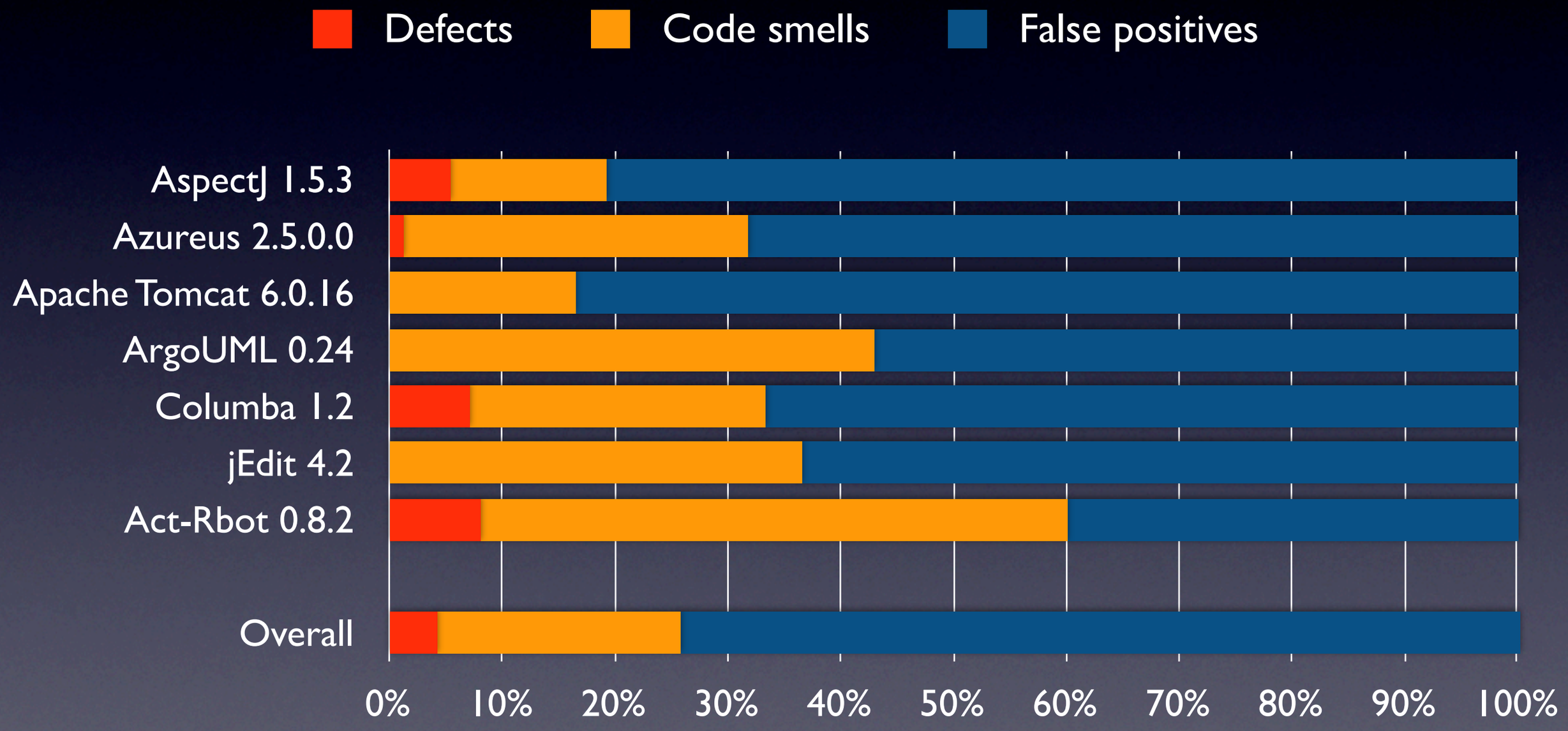
Defects  Code smells  False positives



# AspectJ violations



# More results



# Future work

# Future work

- Procedural languages

# Future work

- Procedural languages
- Interprocedural analysis

# Future work

- Procedural languages
- Interprocedural analysis
- Ranking violations

# Future work

- Procedural languages
- Interprocedural analysis
- Ranking violations
- Early programmer support

# OP-Miner

- OP-Miner learns *operational preconditions*
- Learns from normal argument usage
- Fully automatic
- Found dozens of verified defects