Static Program Checking

CEGAR-based Specification Inference

Automated Software Analysis Group, Institute of Theoretical Informatics

Jun.-prof. Dr. Mana Taghdiri

Thursday– July 17, 2014

Syntactic summaries by abstract interpretation

- Summarizes the behavior of a procedure as a symbolic relationship between pre and post states
- Summaries are declarative formulas in a subset of Alloy
  - Doesn't include quantifiers
  - Doesn't include set comprehension
- Provides both an upper and a lower bound on the final values of fields, return value, and allocated objects

relational expr ⊆ field'/variable' ⊆ relational expr

- The result can sometimes be precise

field'/variable' = relational expr

Evaluations

- To evaluate the quality of the generated summaries
  - Evaluate their accuracy
    - Check if they are sufficient to prove the underlying code correct
- Widening parameters
  - Max number of operators = 1300
  - Mx number of allocations = 5
  - Max number of unions (before closure) = 3
  - Max number of procedure contexts = 5

Accuracy evaluation

- Run on
  - Java linked list
  - An open-source graph library
- All summaries were generated in less than 3 seconds
  - Even for code containing 81 nested distinct method calls
- For each summary, check
  - Summary(proc) => spec(proc)
  - Spec is the actual specification of code, already available
  - The check is done by kodkod, so only in a finite scope
- For 30 procedures
  - 13 had accurate summaries
  - 16 had relatively accurate summaries (sufficient to check major properties)
  - Only in 1 case there was a major loss of information in the summary

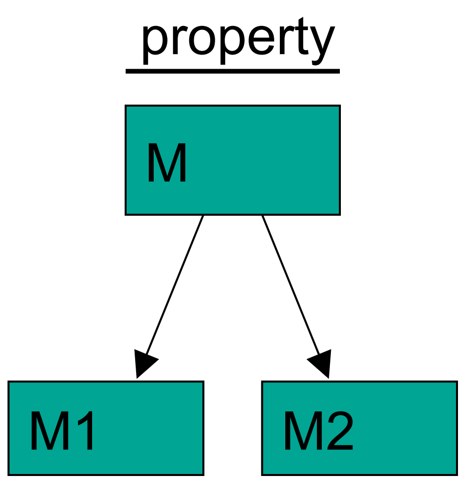
Difficult case

- Remove element from linked list
  - Removes the first occurrence of a given element
  - Although the list is updated at most once, the update is done in the loop
  - So the loop modifies the next field – which is the one traversed in the loop
  - So the loop analysis doesn’t stabilize by inferring closure, but widens to univ
  - The summary allows next and prev fields of all objects to change arbitrarily

CEGAR

- CounterExample-Guided Abstraction Refinement

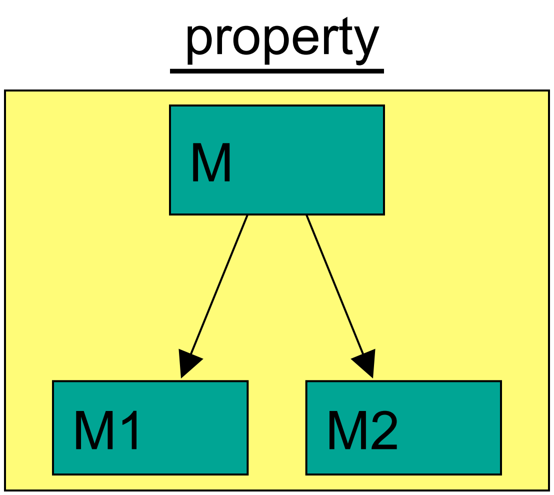
Program verification



Does M satisfy property?

(property is the top-level spec to check)

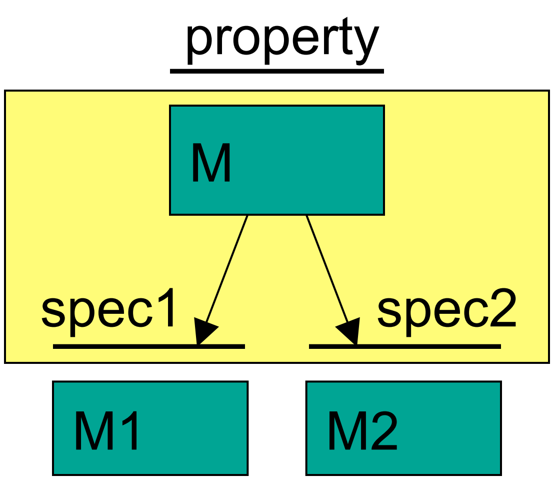
Monolithic program verification



Does M satisfy property?

(considering the code of M1 and M2)

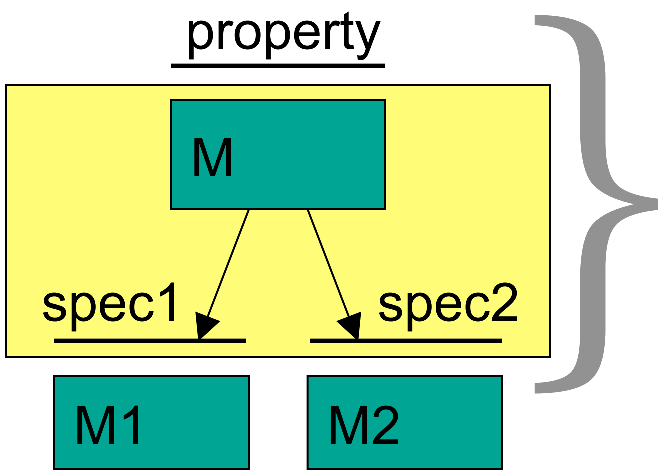
Modular program verification



Does M satisfy property?

(assuming M1 satisfies spec1
and M2 satisfies spec2)

Automating modular verification

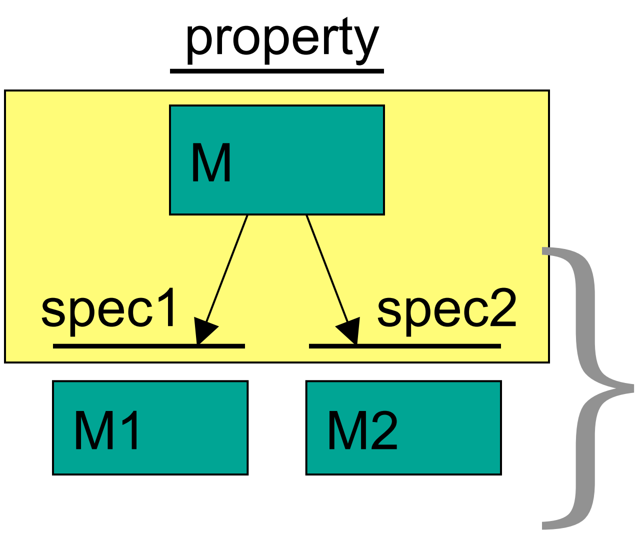


1. How to check the procedure?

Does M satisfy property?

(assuming M1 satisfies spec1
and M2 satisfies spec2)

Automating modular verification



1. How to check the procedure?
2. How to get the specs?

Does M satisfy property?

(assuming M1 satisfies spec1
and M2 satisfies spec2)

Tools we studied

- To check the program:
  - Jalloy
  - Jforge
  - ESC/Java
- To generate specs:
  - Houdini
  - Daikon
  - Static technique for relational specs
- These specs may be insufficient to prove detailed properties

Goal: specs for structural properties

- Target arbitrary data structure properties of code
  - Constrain configuration of objects in the heap
  - Should handle aliasing, reachability, sets, maps, lists, etc.

map

reachability

```
some jw: this.jobsMap'.values'. |
  all x: jw.jobDetail'.listeners'.head'.*next' | some y: job.listeners.head.*next |
    (x.data' = y.data) and #(x.*next') = #(y.*next)
```

quantifiers

set cardinality

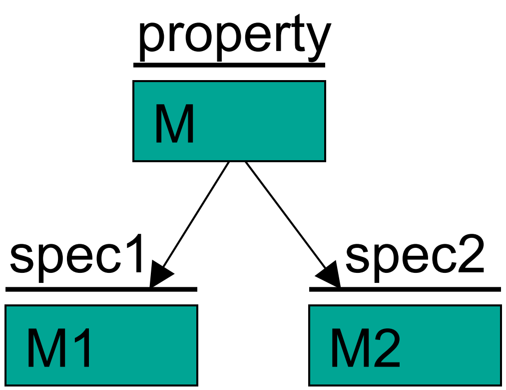
list

```
public void storeJob(JobDetail job,  boolean replaceExisting, …) {
 JobWrapper jw = new JobWrapper(job.clone());
 …
 jobsmap.put(jw.key, jw);
 …
}
```

Insight

Inferring complete specs is infeasible in general

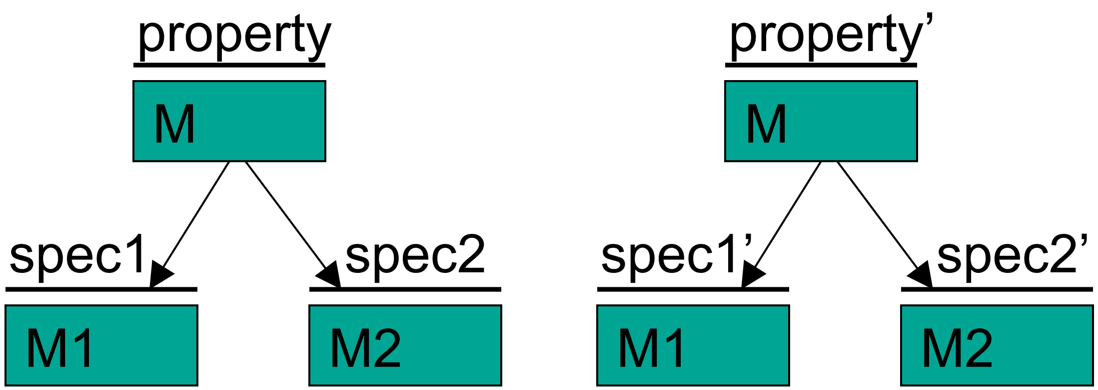
But, necessary specs need not be complete



Insight

Inferring complete specs is infeasible in general

But, necessary specs need not be complete

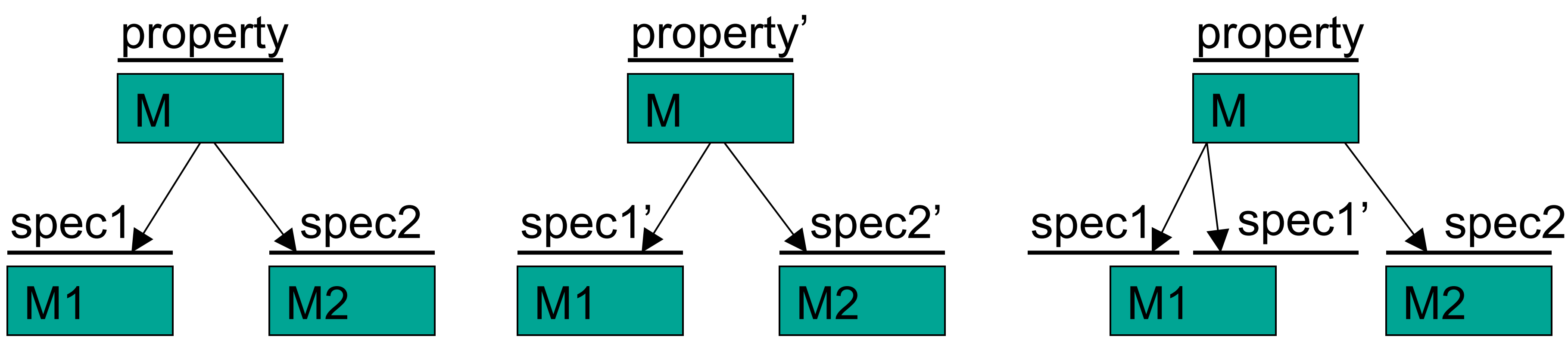


Necessary specs are property dependent

Insight

Inferring complete specs is infeasible in general

But, necessary specs need not be complete



Necessary specs are property dependent

Necessary specs are call site dependent

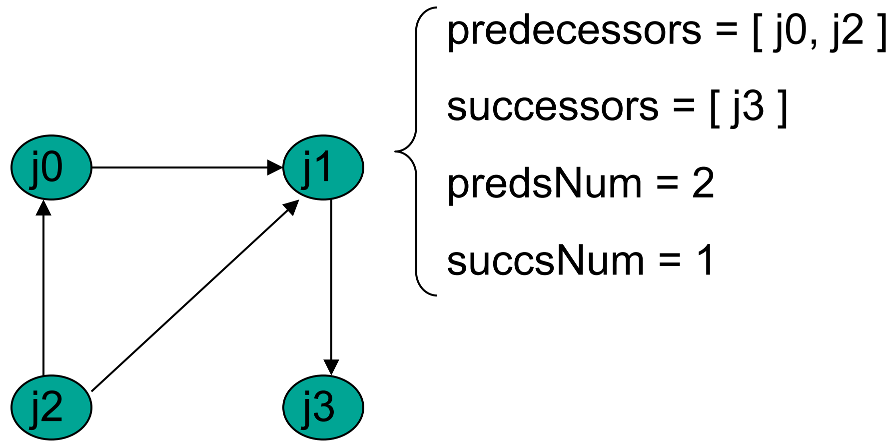
Example

```
class Job {
   JobList predecessors;
   JobList successors;
   int predsNum;
   int succsNum;
   int visitedPredsNum;
}
class JobList {
   Entry head;
}
class Entry {
   Job job;
   Entry next;
}
```

```
boolean scheduleJobs(JobList l) {
      boolean isAcyclic = true;
      l.init();
      Entry cur = l.head;
      while (cur != null) {
         Entry ready = findReady(cur);
         if (ready == null) {
            isAcyclic = false; break;  }
         fixVisited(ready.job);
         swapJobs(ready, cur);
         cur = cur.next;
      } return isAcyclic;
}
```

Example

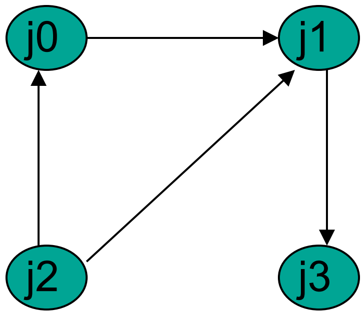
```
class Job {
  JobList predecessors;
  JobList successors;
  int predsNum;
  int succsNum;
  int visitedPredsNum;
}
class JobList {
  Entry head;
}
class Entry {
  Job job;
  Entry next;
}
```



Example

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```

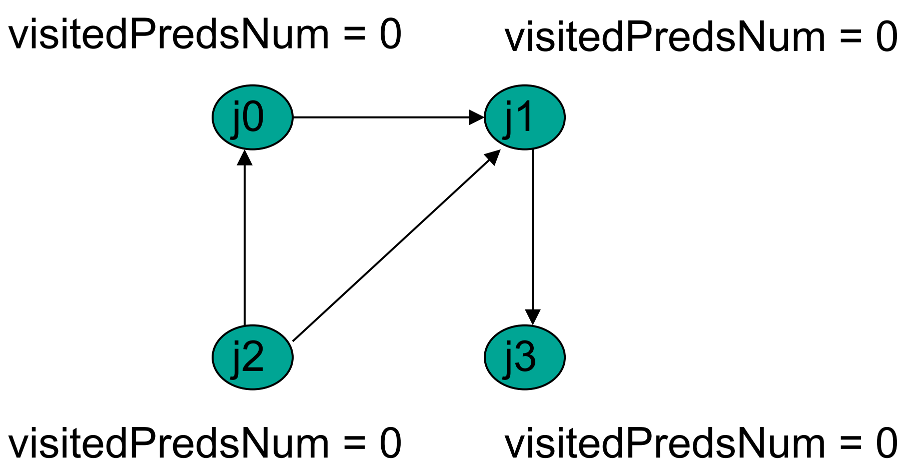
l = [j0, j1, j2, j3]



Example

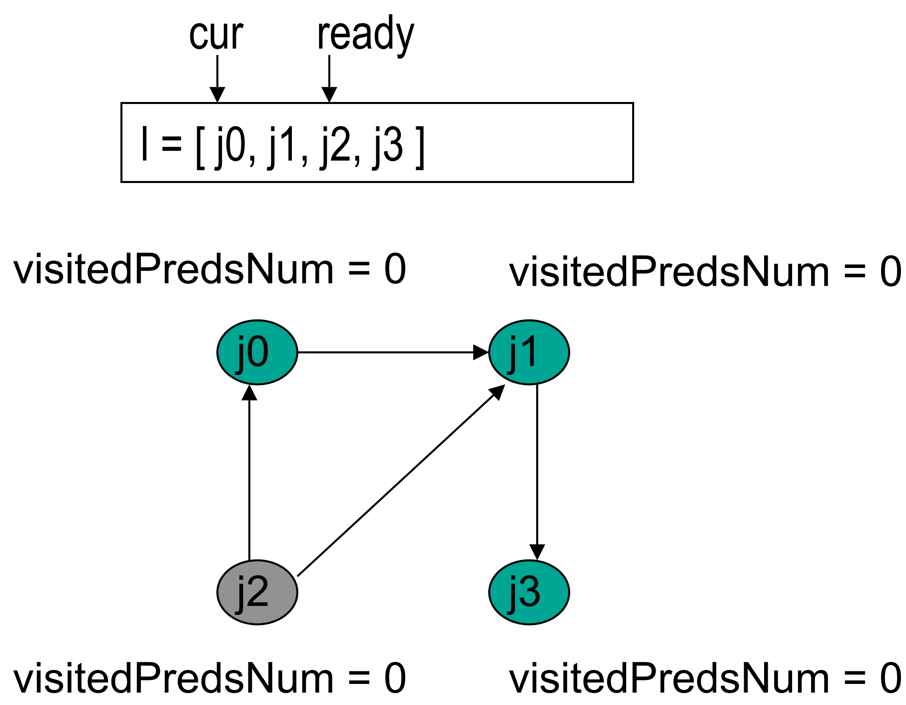
```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```

l = [j0, j1, j2, j3]



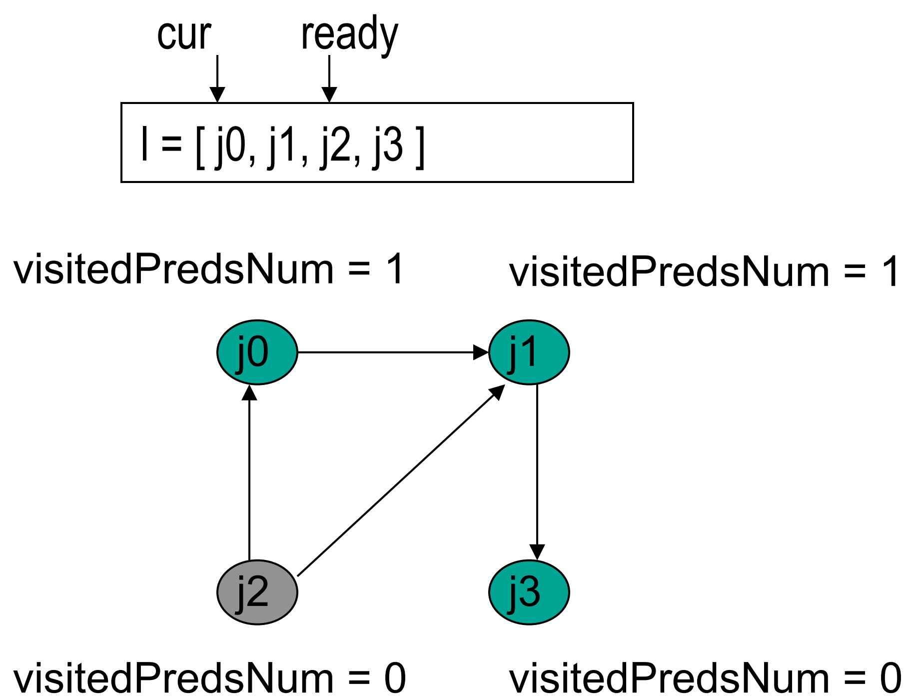
Example

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```



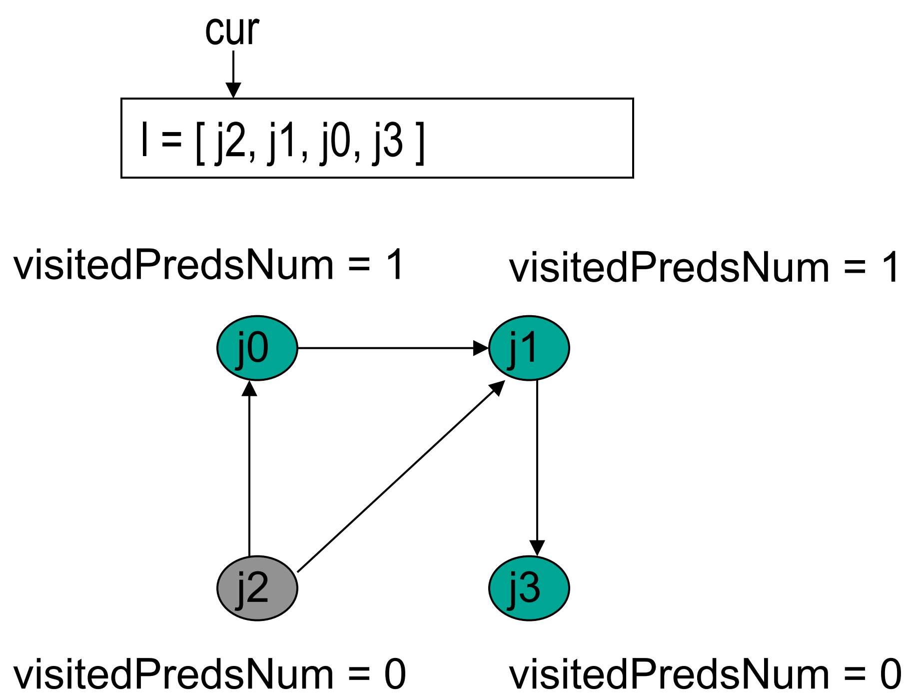
Example

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```



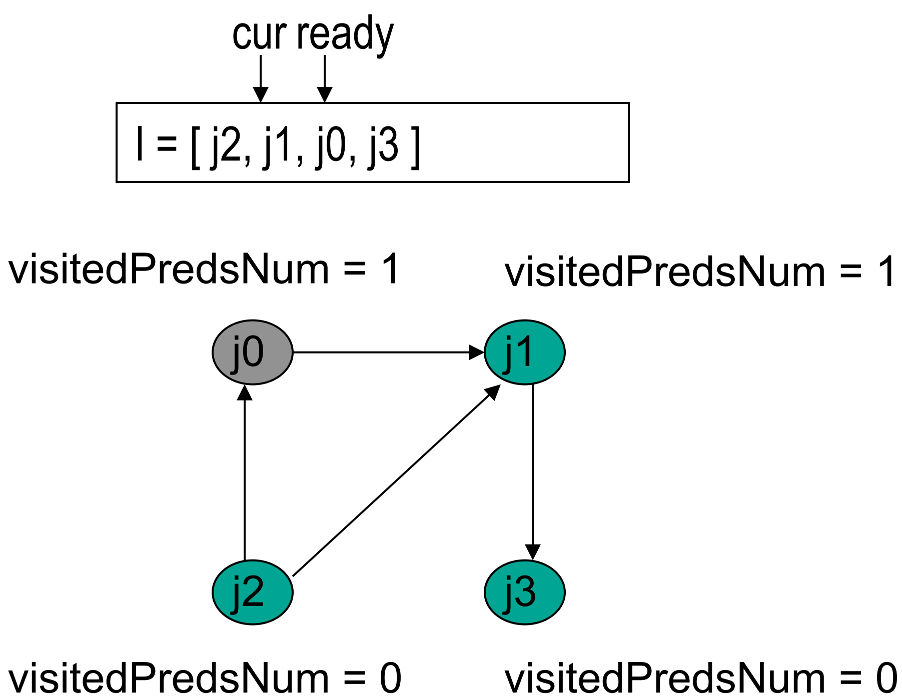
Example

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```



Example

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```



Example

```
class Job {
  JobList predecessors;
  JobList successors;
  int predsNum;
  int succsNum;
  int visitedPredsNum;
}
class JobList {
  Entry head;
}
class Entry {
  Job job;
  Entry next;
}
```

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```

jobs = head.*next.job

Eventually schedules all jobs (l.jobs' = l.jobs)

Example – call site specs

(l.jobs’ = l.jobs)

Property can be checked with full specs for all call sites

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```

full spec:
For all jobs, initializes its “visitedPredsNum” to 0

full spec:
Returns first node reachable from “cur” whose “visitedPredsNum” equals “predsNum”

full spec:
Increments “visitedPredsNum” of all successors of “e.job”

full spec:
Swaps the jobs of the given entries

Example – call site specs

(l.jobs' = l.jobs)

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```

???

???

???

???

Property can be checked with full specs for all call sites

But, for some calls, partial specs are enough

Example – call site specs

(l.jobs’ = l.jobs)

Property can be checked with full specs for all call sites

But, for some calls, partial specs are enough

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```

partial spec:
jobs’ = jobs

partial spec:
$ret ⊆ cur.*next

partial spec:
jobs’ = jobs

full spec:
Swaps the jobs of the given entries

Example – call site specs

(l.jobs' = l.jobs)

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break;  }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```

partial spec:
jobs' = jobs

partial spec:
$ret $\subseteq$ cur.*next

partial spec:
jobs' = jobs

full spec:
Swaps the jobs of the given entries

Property can be checked with full specs for all call sites

But, for some calls, partial specs are enough

We can infer these specs automatically

Approach: infer sufficient specs

Benefits

- Analyzes only as much code as necessary
- Often performs better than inlining
  (reduces the analysis time of the example by factor of 15)
- Finds callees' bugs if relevant to the property

Compromises

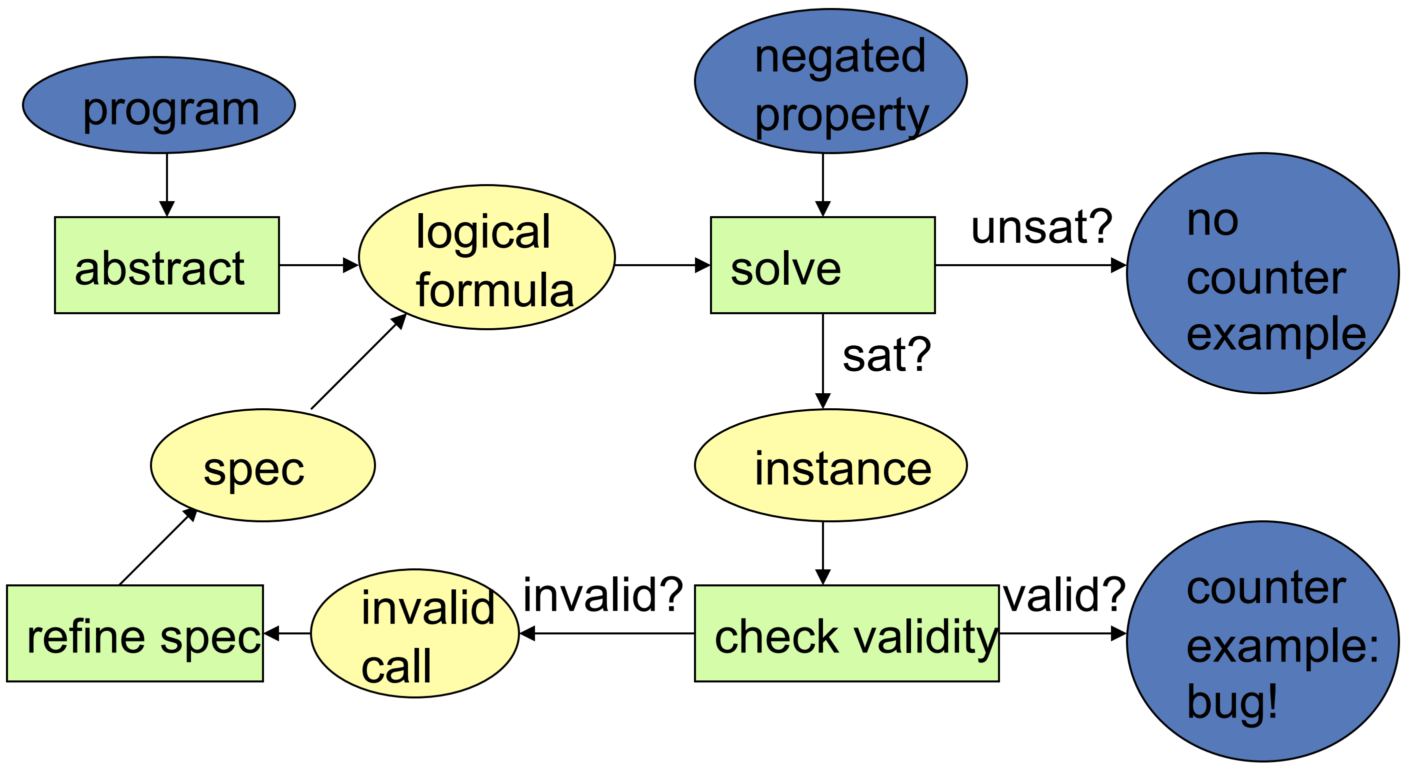
- Finitizes heap
- Finitizes loops/recursions

Is a bug finder

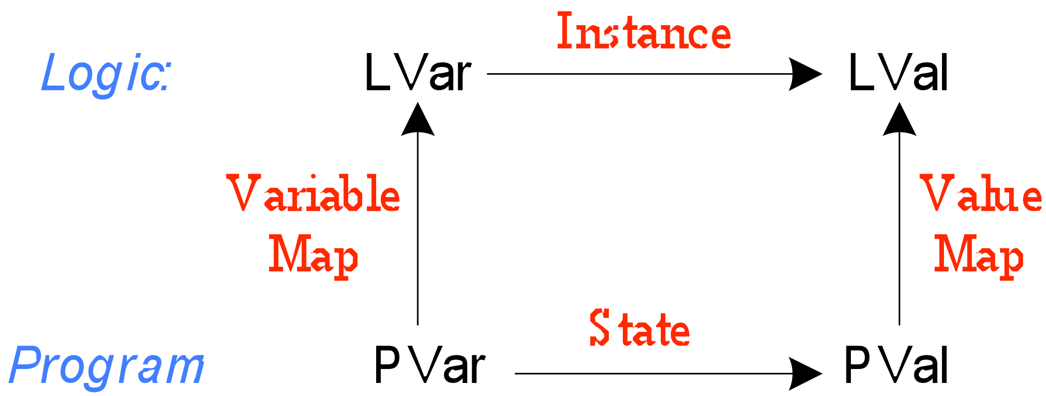
- Sound (no false negatives)
- Complete within bounds (bounded verification)

Algorithmic overview: CEGAR

specification refinement



Framework: domains



- Logic domain:
  - LVar (logical variables), LVal (logical values)
- Program domain:
  - PVar (program variables), PVal (program values)

Framework: input functions

translate: Stmt × VarMap × ValueMap → Formula × VarMap

- Translates code to finite formula

solve: Formula × Instance → $\mathcal{P}$Instance
solve(f, i) = { i' | ($i \subseteq i'$) ∧ ($i' \in [\![f]\!]$) }

- Solves a formula with respect to a partial instance

invalidate: Formula × Instance → Formula
(invalidate(f, i) = g) ⇒ (solve(g, i) = ∅) ∧ ($f \Rightarrow g$)

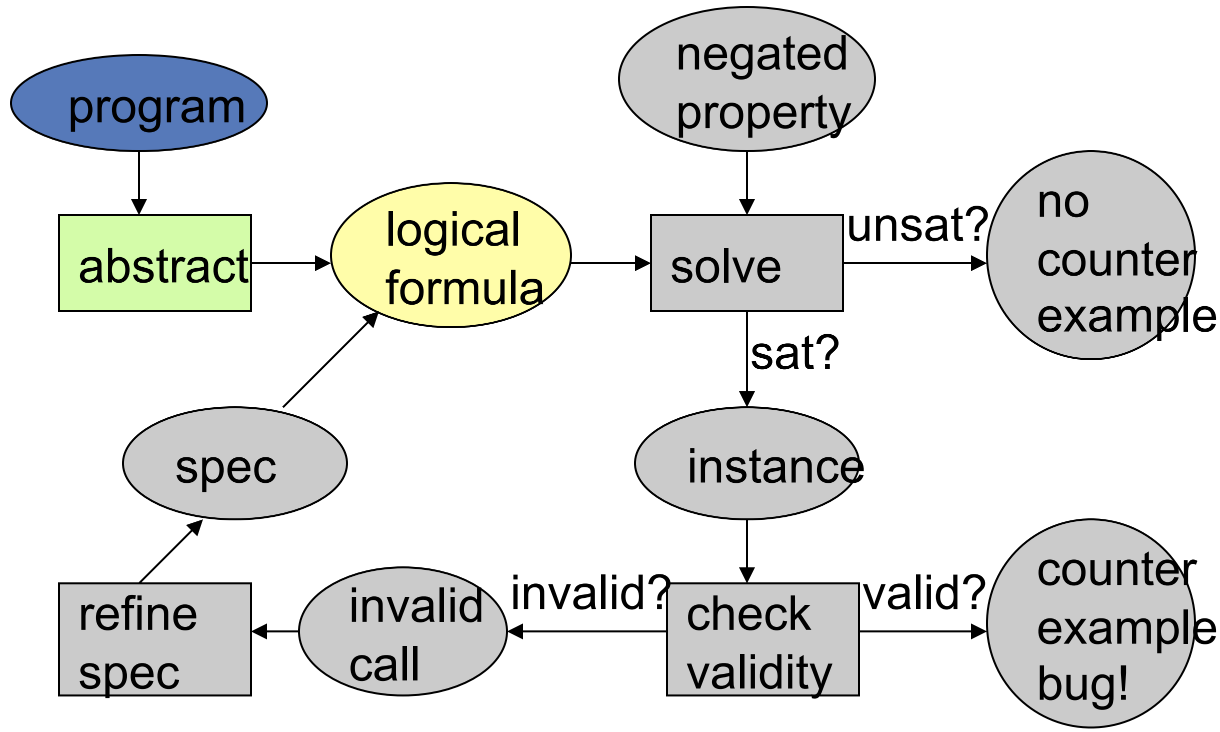
- Generates a formula that invalidates an instance

spec: CallStmt × VarMap × ValueMap → Formula × VarMap

- Overapproximates the effects of a call site

Framework: abstraction

procedure p() {
  stmt1;
  ~~q();~~ spec(q)
  stmt2;
}



- Translates all statements but call sites
- Replaces call sites with their specs
  - Empty specs
  - Frame condition
  - Richer specs

Example – relational view

Relational view of the heap:

- Types: sets
- Fields: binary functional relations
- Variables: singleton sets

```
class JobList {
  Entry head; }

class Entry {
  Job job;
  Entry next; }

class Job {
  JobList predecessors;
  JobList successors;
  int predsNum;
  int succsNum;
  int visitedPredsNum; }
```

JobList, Entry, Job : set Obj

head: JobList → Entry

job: Entry → Job

next: Entry → Entry

predecessors: Job → JobList

successors: Job → JobList

predsNum: Job → Int

succsNum: Job → Int

visitedPredsNum: Job → Int

Example – empty specs

Karlsruhe Institute of Technology

```
Entry findReady(Entry e) {
  Entry c = e;
  while ((c != null) &&
           (c.job.predsNum != c.job.visitedPredsNum))
     c = c.next;
  return c;
}
```

return = $?_{Entry}$
job' = ?
next' = ?
head' = ?
predecessors' = ?
…

```
void fixVisited(Job j) {
  Entry e = j.successors.head;
  while (e != null) {
     e.job.visitedPredsNum =  e.job.visitedPredsNum + 1;
     e = e.next;
  }
}
```

visitedPredsNum' = $?_{Job} \rightarrow ?_{Int}$
job' = ?
next' = ?
head' = ?
…

Example – frame conditions

```
Entry findReady(Entry e) {
  Entry c = e;
  while ((c != null) &&
          (c.job.predsNum != c.job.visitedPredsNum))
     c = c.next;
  return c;
}
```

??

```
void fixVisited(Job j) {
  Entry e = j.successors.head;
  while (e != null) {
     e.job.visitedPredsNum =  e.job.visitedPredsNum + 1;
     e = e.next;
  }
}
```

??

Example – frame conditions

```
Entry findReady(Entry e) {
  Entry c = e;
  while ((c != null) &&
           (c.job.predsNum != c.job.visitedPredsNum))
     c = c.next;
  return c;
}
```

return = $?_{Entry}$
job' = job
next' = next
head' = head
predecessors' = predecessors
…

```
void fixVisited(Job j) {
  Entry e = j.successors.head;
  while (e != null) {
    e.job.visitedPredsNum =  e.job.visitedPredsNum + 1;
    e = e.next;
  }
}
```

visitedPredsNum' = $?_{Job} \rightarrow ?_{Int}$
job' = job
next' = next
head' = head
…

Example – abstraction

(l.jobs' = l.jobs)

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    ~~l.init();~~
    Entry cur = l.head;
    while (cur != null) {
      ~~Entry ready = findReady(cur);~~
      if (ready == null) {
        isAcyclic = false; break;  }
      ~~fixVisited(ready.job);~~
      ~~swapJobs(ready, cur);~~
      cur = cur.next;
    } return isAcyclic;
}
```

visitedPredsNum' = $?_{Job} \rightarrow ?_{Int}$

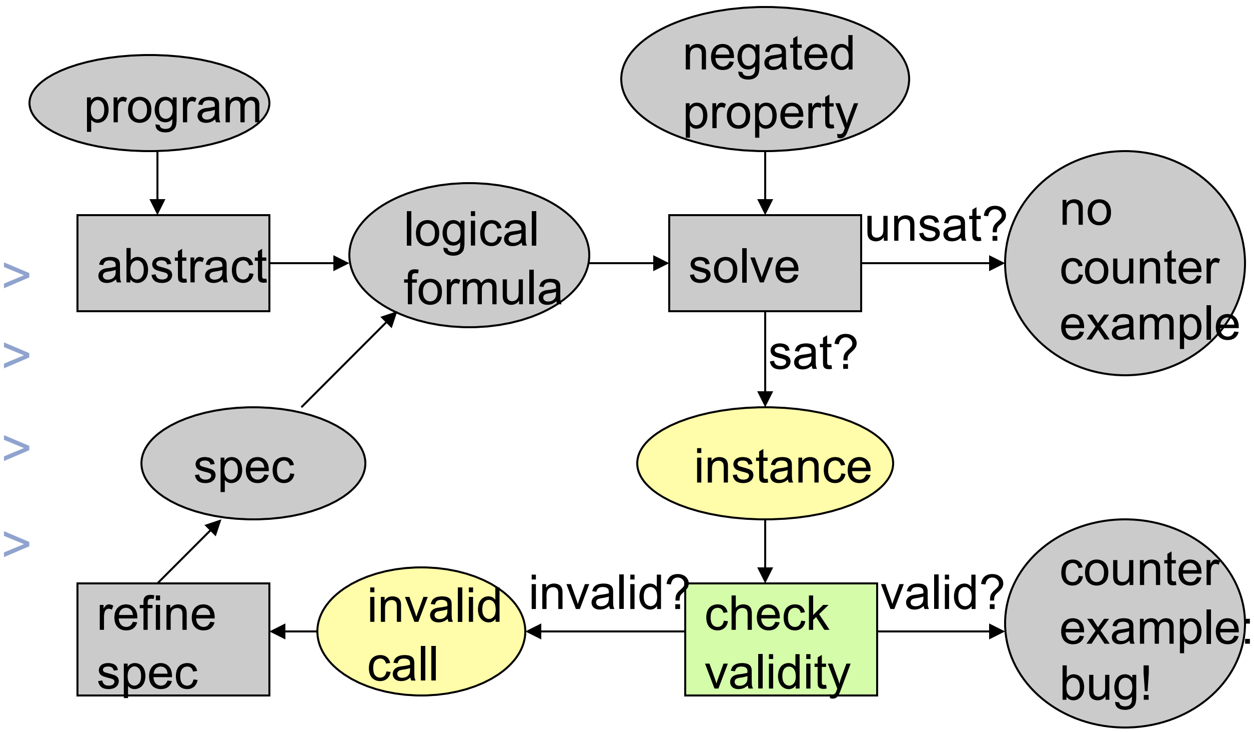
\$ret = $?_{Entry}$

visitedPredsNum' = $?_{Job} \rightarrow ?_{Int}$

job' = $?_{Entry} \rightarrow ?_{Job}$

Framework: validity check

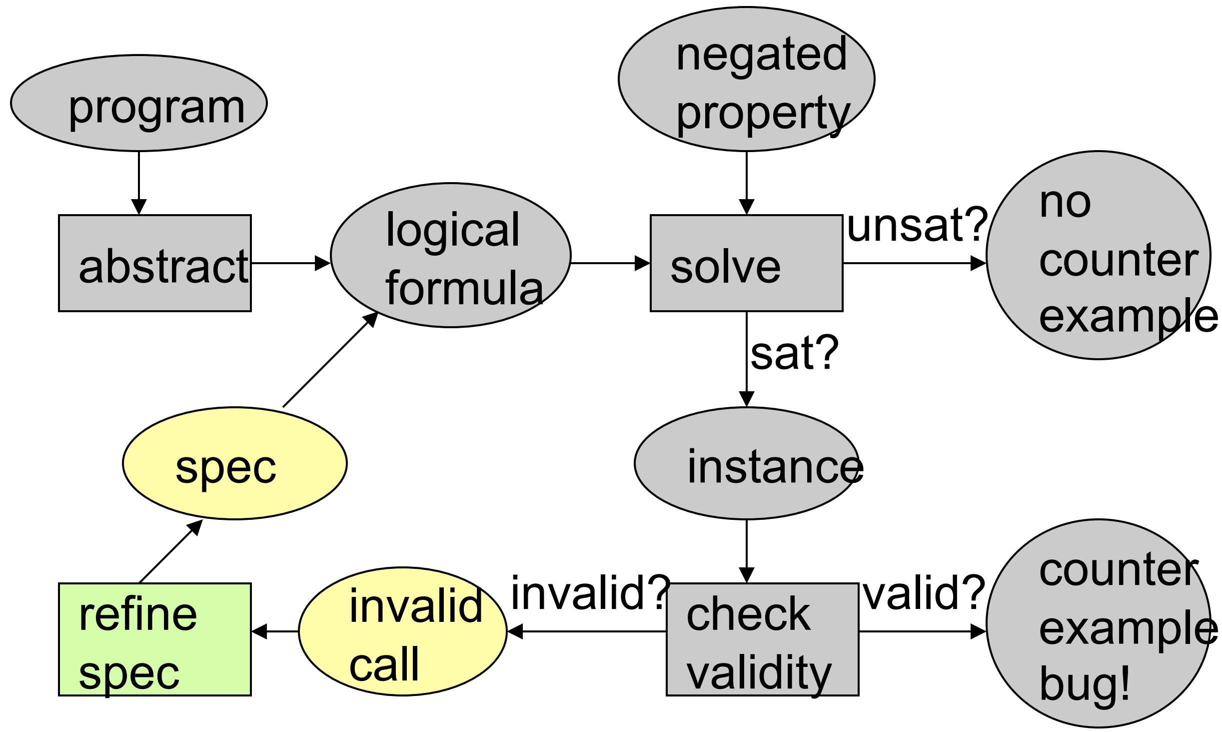
procedure p() {
<S0>
stmt1;
<S1>
~~q();~~ spec(q)
<S2>
stmt2;
<S3>
}



- Instance = trace in abstract program
- Examine each call: solve
  S1 ∧ translate(q) ∧ S2
- If unsatisfiable, the call is invalid

Framework: spec refinement

procedure p() {
  stmt1; ?
  ~~q(); spec(q)~~
  stmt2;
}

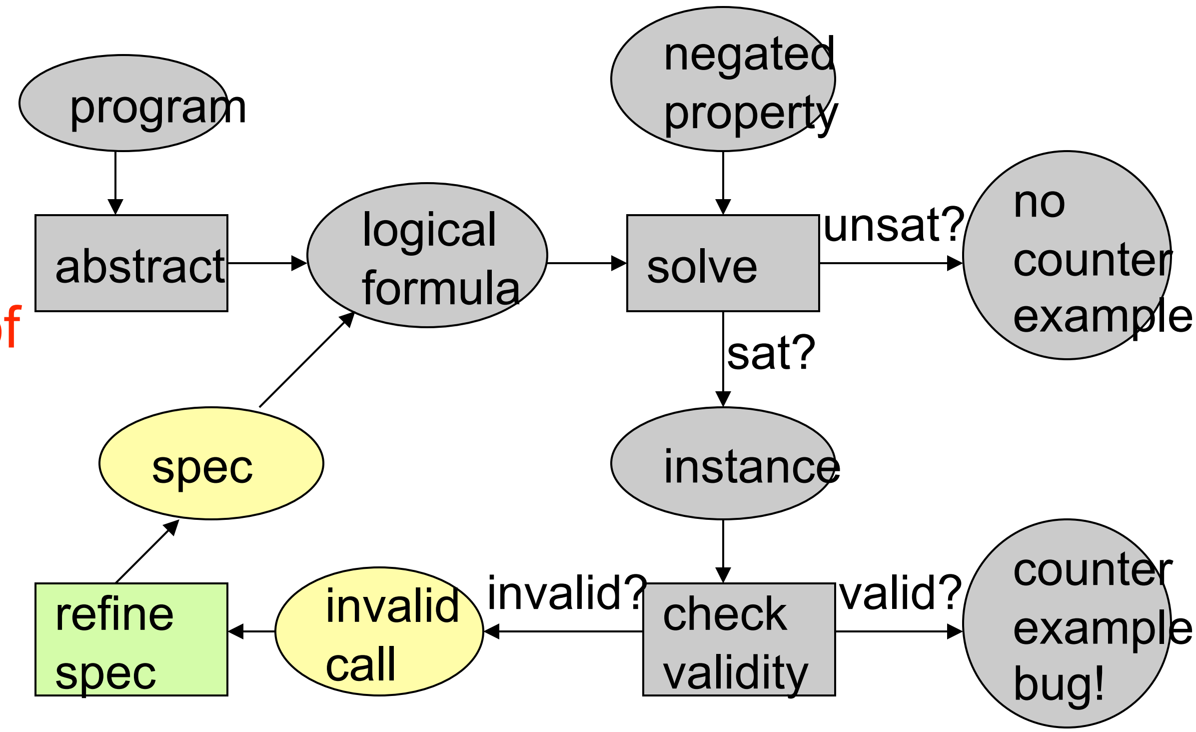


preState(q) ∧ translate(q) ∧ postState(q) = false

- New spec must eliminate (preState(q), postState(q))
- translate(q) can be the new spec
- Too big!

Framework: spec refinement

procedure p() {
  stmt1; spec(q) ∧ proof
  ~~q(); spec(q)~~
  stmt2;
}



preState(q) ∧ translate(q) ∧ postState(q) = false

- Proof of unsatisfiability:

An unsatisfiable consequence of the solved formula

translate(q) => proof

preState(q) ∧ proof ∧ postState(q) = false

Example – spec refinement

(l.jobs' = l.jobs)

```
boolean scheduleJobs(JobList l) {
    boolean isAcyclic = true;
    l.init();
    Entry cur = l.head;
    while (cur != null) {
      Entry ready = findReady(cur);
      if (ready == null) {
        isAcyclic = false; break; }
      fixVisited(ready.job);
      swapJobs(ready, cur);
      cur = cur.next;
    } return isAcyclic;
}
```

(cur = E0)
(E0.job = J0) (E0.next = null)
…

Pre-state

(ready = E1)
(E1.job = J1) (E1.next = null)
…

Post-state

Why is this a possible instance???

Spec refinement – example

```
Entry findReady(Entry e) {
  Entry c = e;
  while ((c != null) &&
    (c.job.predsNum != c.job.visitedPredsNum))
      c = c.next;
  return c;
}
```

Pre-state

(e = E0)
(E0.job = J0) (E0.next = null)
…

Post-state

($ret = E1)
(E1.job = J1) (E1.next = null)
…

Translation of findReady (one unrolling)

(e = null) ⇒ ($ret = e)

((e != null) ∧ (e.job.predsNum = e.job.visitedPredsNum))
⇒ ($ret = e)

((e != null) ∧ (e.job.predsNum != e.job.visitedPredsNum))
⇒ ($ret = e.next)

(e.next = null)

Spec refinement – example

```
Entry findReady(Entry e) {
  Entry c = e;
  while ((c != null) &&
    (c.job.predsNum != c.job.visitedPredsNum))
      c = c.next;
  return c;
}
```

Pre-state

(e = E0)
(E0.job = J0) (E0.next = null)
…

Post-state

($ret = E1)
(E1.job = J1) (E1.next = null)
…

Translation of findReady
(one unrolling)

$(e = null) \Rightarrow (\$ret = e)$

$((e \mathrel{!=} null) \wedge (e.job.predsNum = e.job.visitedPredsNum)) \Rightarrow (\$ret = e)$

$((e \mathrel{!=} null) \wedge (e.job.predsNum \mathrel{!=} e.job.visitedPredsNum)) \Rightarrow (\$ret = e.next)$

$(e.next = null)$

---

false

Spec refinement – example

KIT Karlsruhe Institute of Technology

```
Entry findReady(Entry e) {
  Entry c = e;
  while ((c != null) &&
    (c.job.predsNum != c.job.visitedPredsNum))
      c = c.next;
  return c;
}
```

Pre-state

(e = E0)
(E0.job = J0) (E0.next = null)
…

Post-state

($ret = E1)
(E1.job = J1) (E1.next = null)
…

Translation of findReady (one unrolling)

(e = null) ⇒ ($ret = e)

((e != null) ∧ (e.job.predsNum = e.job.visitedPredsNum)) ⇒ ($ret = e)

((e != null) ∧ (e.job.predsNum != e.job.visitedPredsNum)) ⇒ ($ret = e.next)

(e.next = null)

Spec refinement – example

```
Entry findReady(Entry e) {
  Entry c = e;
  while ((c != null) &&
    (c.job.predsNum != c.job.visitedPredsNum))
      c = c.next;
  return c;
}
```

Pre-state

(e = E0)
(E0.job = J0) (E0.next = null)
…

Post-state

($ret = E1)
(E1.job = J1) (E1.next = null)
…

Translation of findReady (one unrolling)

(e = null) ⇒ ($ret = e)

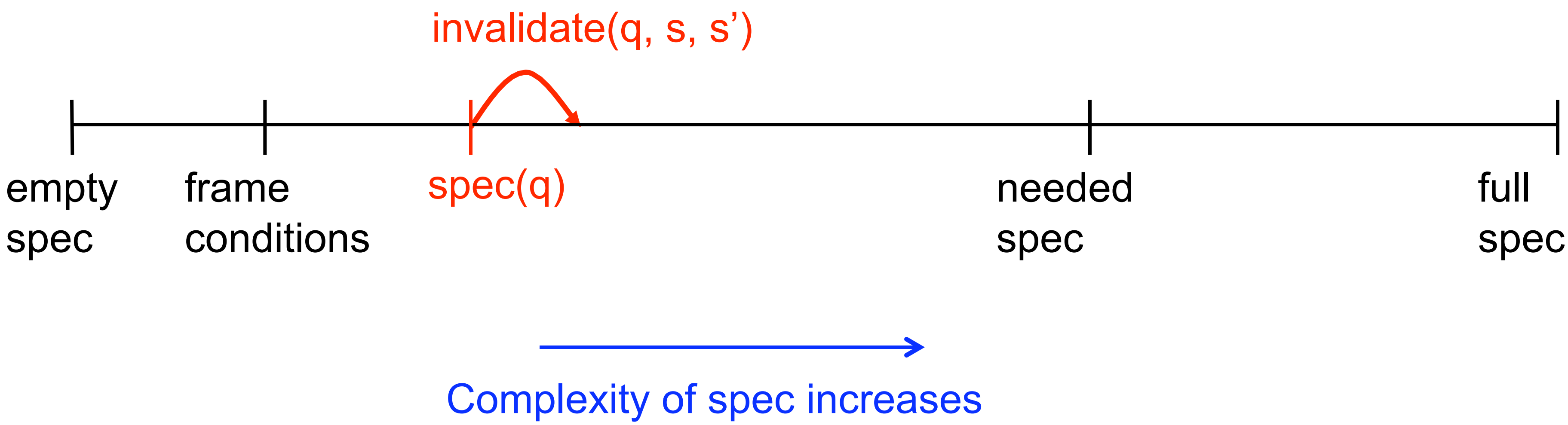
((e != null) ∧ (e.job.predsNum = e.job.visitedPredsNum))
⇒ ($ret = e)

((e != null) ∧ (e.job.predsNum != e.job.visitedPredsNum))
⇒ ($ret = e.next)

(e.next = null)

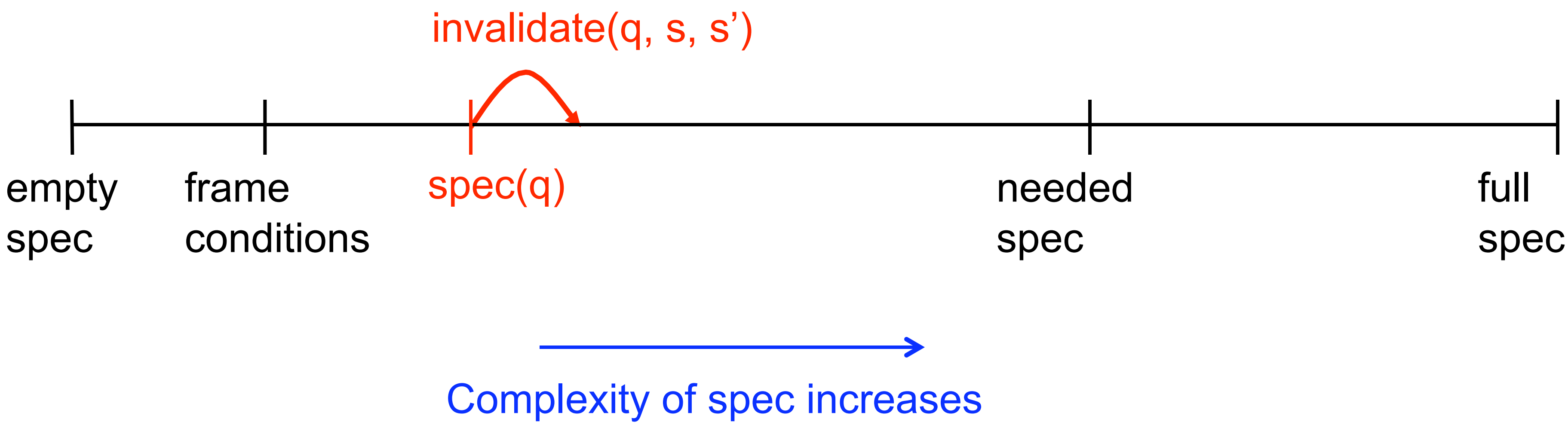
New spec:
($ret = e) || ($ret = e.next)

Spec refinement



- What is a good spec to start with?
  - Are frame conditions good enough?
- What is a good pace to make progress?
  - Is proof of unsatisfiability good enough?

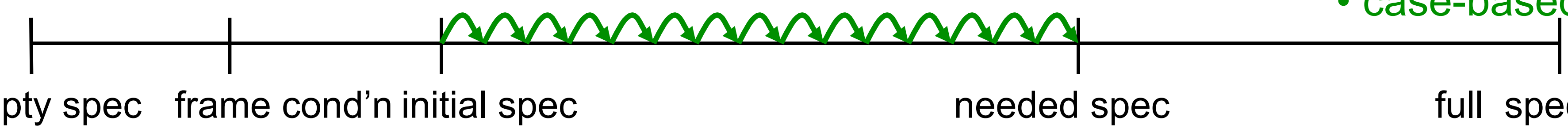
Spec refinement



- Other ideas for 'invalidate'?
  - Translate(q) and S and S' = False

Spec refinement: pace of progress

• case-based



pre-state (s)

procedure q(..) {

.

.

.

}

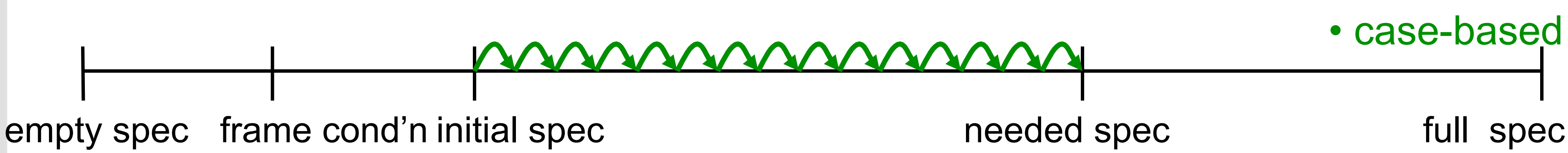
post-state (s')

case-based:

$\neg(s \wedge s')$

• good or bad?

Spec refinement: pace of progress



pre-state (s)

procedure q(..) {

.

.

.

}

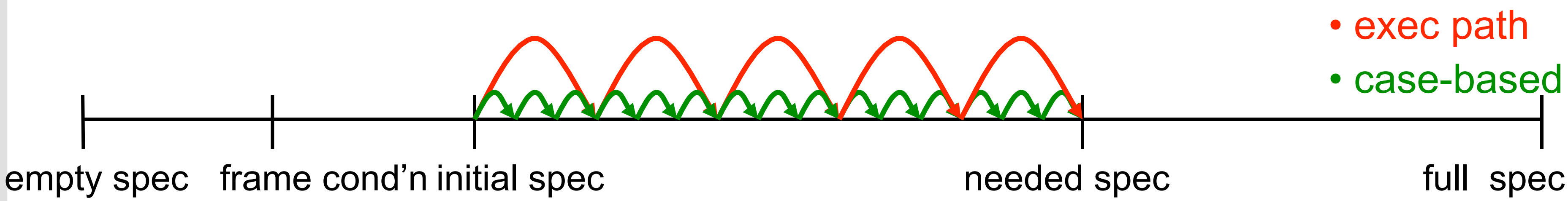
post-state (s')

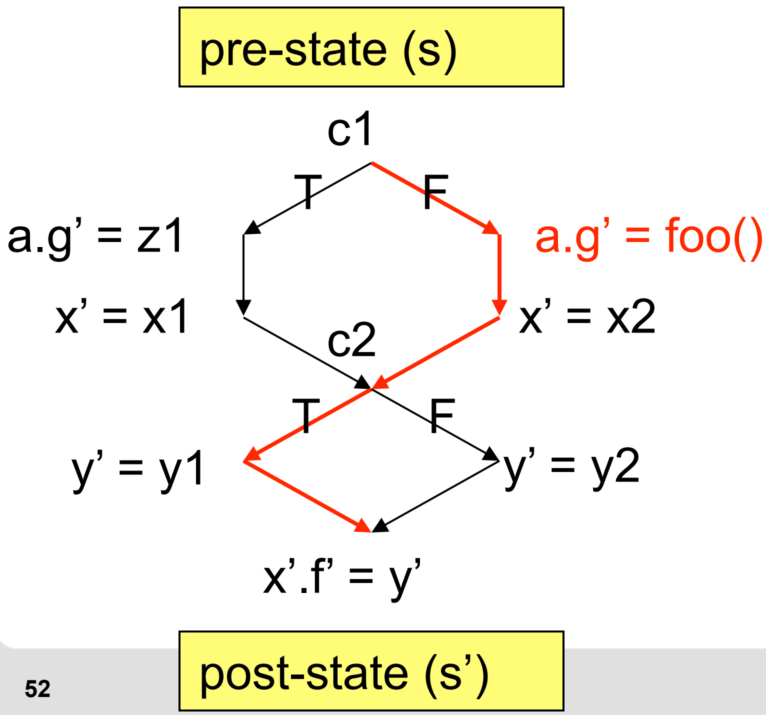
case-based:

$\neg(s \land s')$

- number of refinements proportional to the number of states allowed

Spec refinement: pace of progress





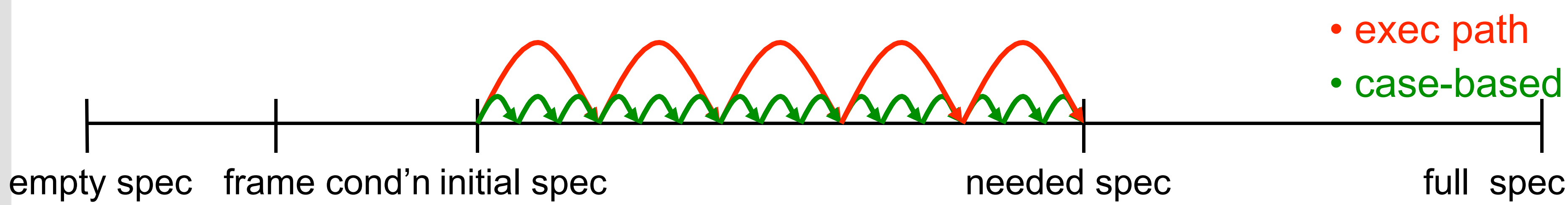
exec path:

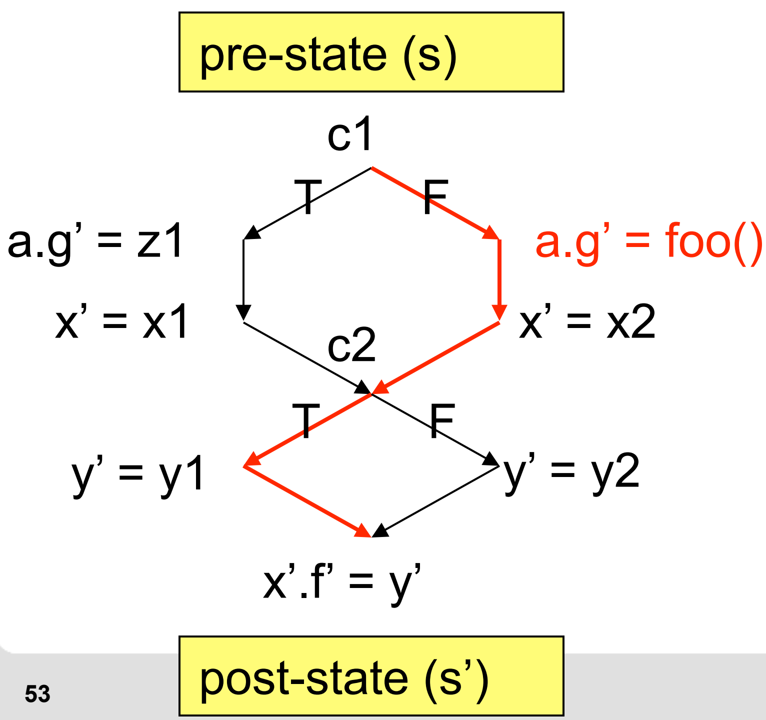
$(\neg c1 \wedge c2) \Rightarrow f' = f \mathbin{++} x2 \rightarrow y1 \quad \wedge$

$g' = g \mathbin{++} a \rightarrow$ foo_ret

• pros / cons?

Spec refinement: pace of progress





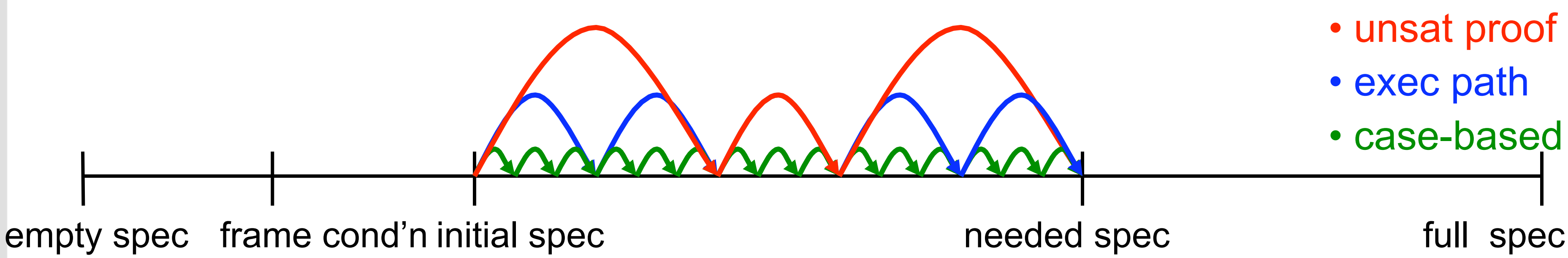
exec path:

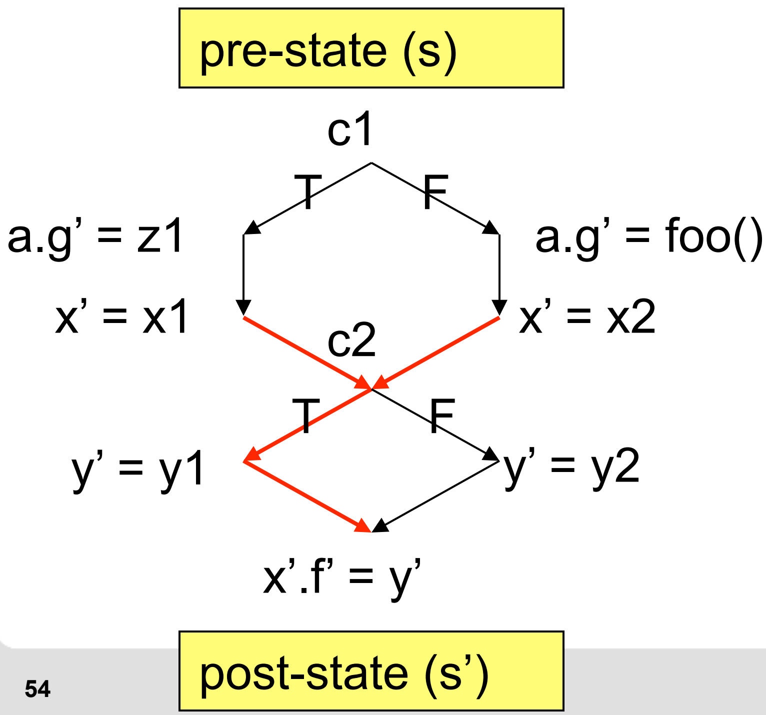
$(\neg c1 \wedge c2) \Rightarrow f' = f \mathbin{++} x2 \rightarrow y1 \quad \wedge$

$\qquad g' = g \mathbin{++} a \rightarrow \text{foo\_ret}$

- number of refinements proportional to the number of paths
- refined spec may be unnecessarily complex

Spec refinement: pace of progress





unsat proof:

$c2 \Rightarrow f' = (f \mathbin{++} x1 \rightarrow y1) \;\|$

$\quad f' = (f \mathbin{++} x2 \rightarrow y1)$

- extracts only relevant pieces
- may encode more than one path
- checks an abstraction of code: inner calls still abstract

Initial specifications

Affects performance

- Better specs reduce number of refinements
- Time spent to get rich specs may be wasted

Lightweight technique:

- Specifies upper and lower bounds on final values

relational expr ⊆ field'/variable' ⊆ relational expr

- Results are sometimes precise

field'/variable' = relational expr

Example – relational specs

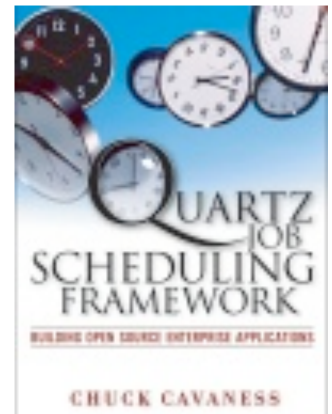
```
Entry findReady(Entry e) {
  Entry c = e;
  while ((c != null) &&
            (c.job.predsNum != c.job.visitedPredsNum))
     c = c.next;
  return c;
}
```

$\$ret \subseteq (e.^*next \ \&\ (null + job.predsNum.(e.^*next.job.visitedPredsNum)))$

$\$ret \supseteq \varnothing$

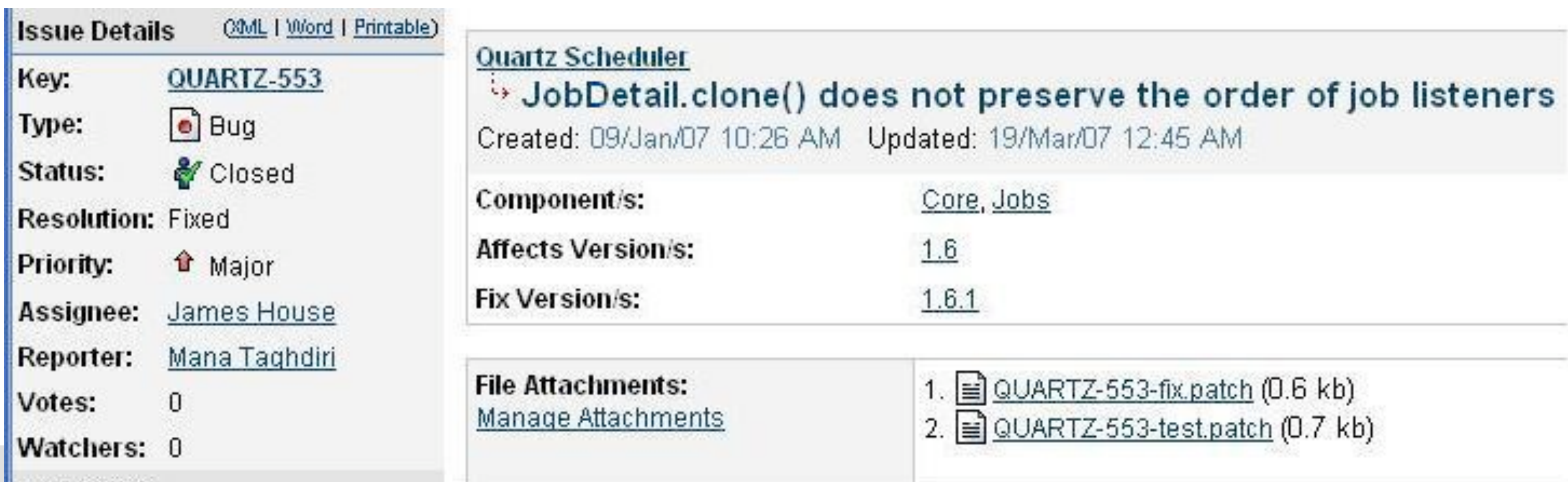
Experiments – Quartz API

- Open source library for job scheduling
  - Used by “thousands of people”
  - Uses several data structures (e.g. list, map, set, ordered set)
- Checked 40 Methods in 4 Units
  - Containing up to 53 distinct called methods
- Checked correctness properties
  - Extracted from comments
  - Written as partial specs
- Found two previously unknown bugs



1st bug found in Quartz

- The bug:
  - In a called procedure (a very basic one)
  - Observable by users of Quartz
- This particular post-condition was never tested by developers
- The code:
  - Contains 17 method calls
  - Accesses 4 different maps and 2 ordered sets



2nd bug found in Quartz

- The bug:
  - In a called procedure (one of the overriding ones)
  - Observable by users of Quartz
- This particular path was not covered by any unit test
- The code:
  - Requires dynamic dispatch
  - Contains 28 method calls
  - Accesses 1 map and 2 lists

Issue Details (XML | Word | Printable)

Key:	QUARTZ-557
Type:	Bug
Status:	Closed
Resolution:	Fixed
Priority:	Major
Assignee:	Henri Yandell
Reporter:	Mana Taghdiri
Votes:	1
Watchers:	1

Quartz Scheduler

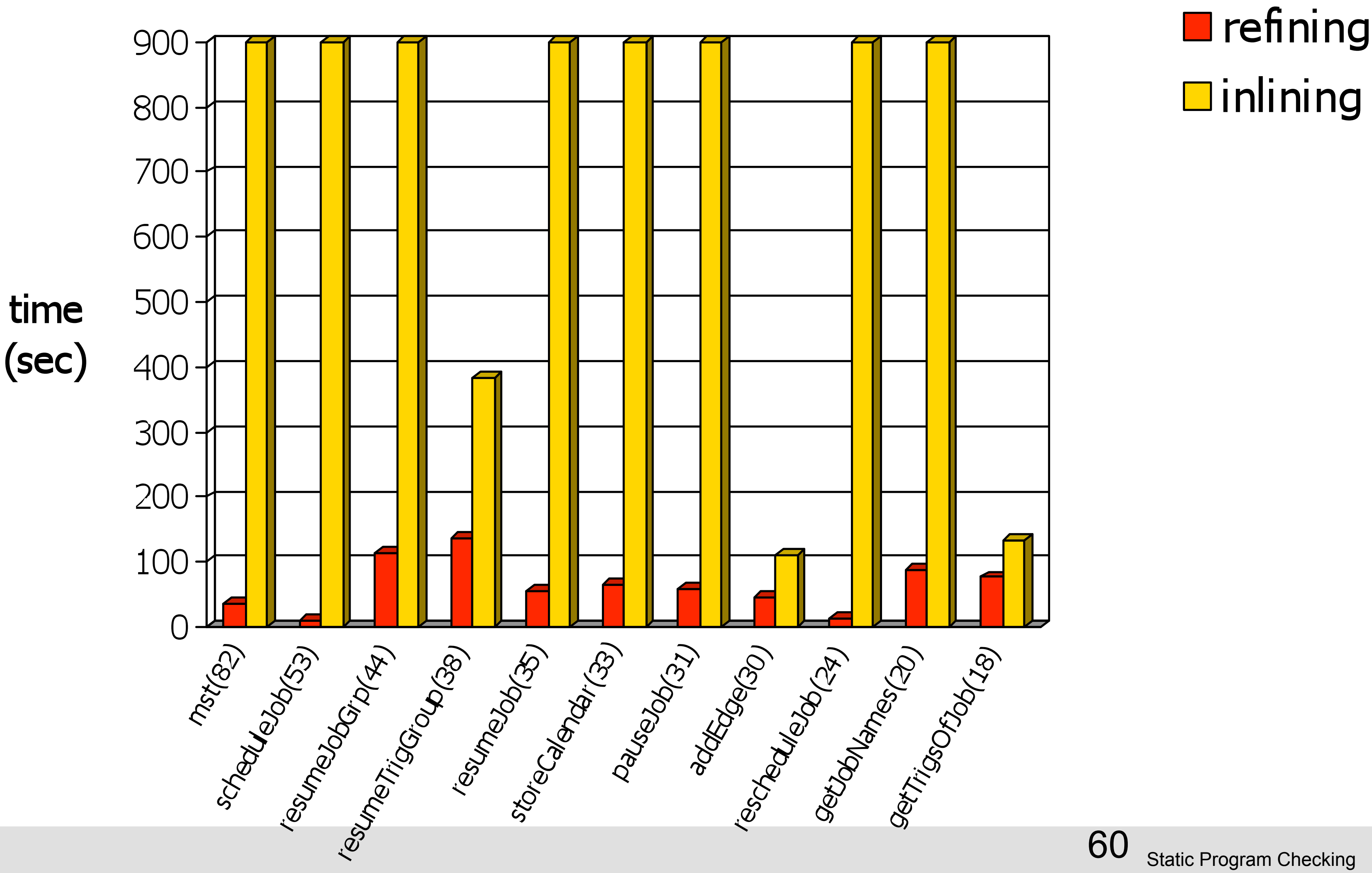
Cron/Simple Trigger may return a firing time not included in the calender

Created: 26/Jan/07 12:43 PM Updated: 06/Sep/07 02:16 PM

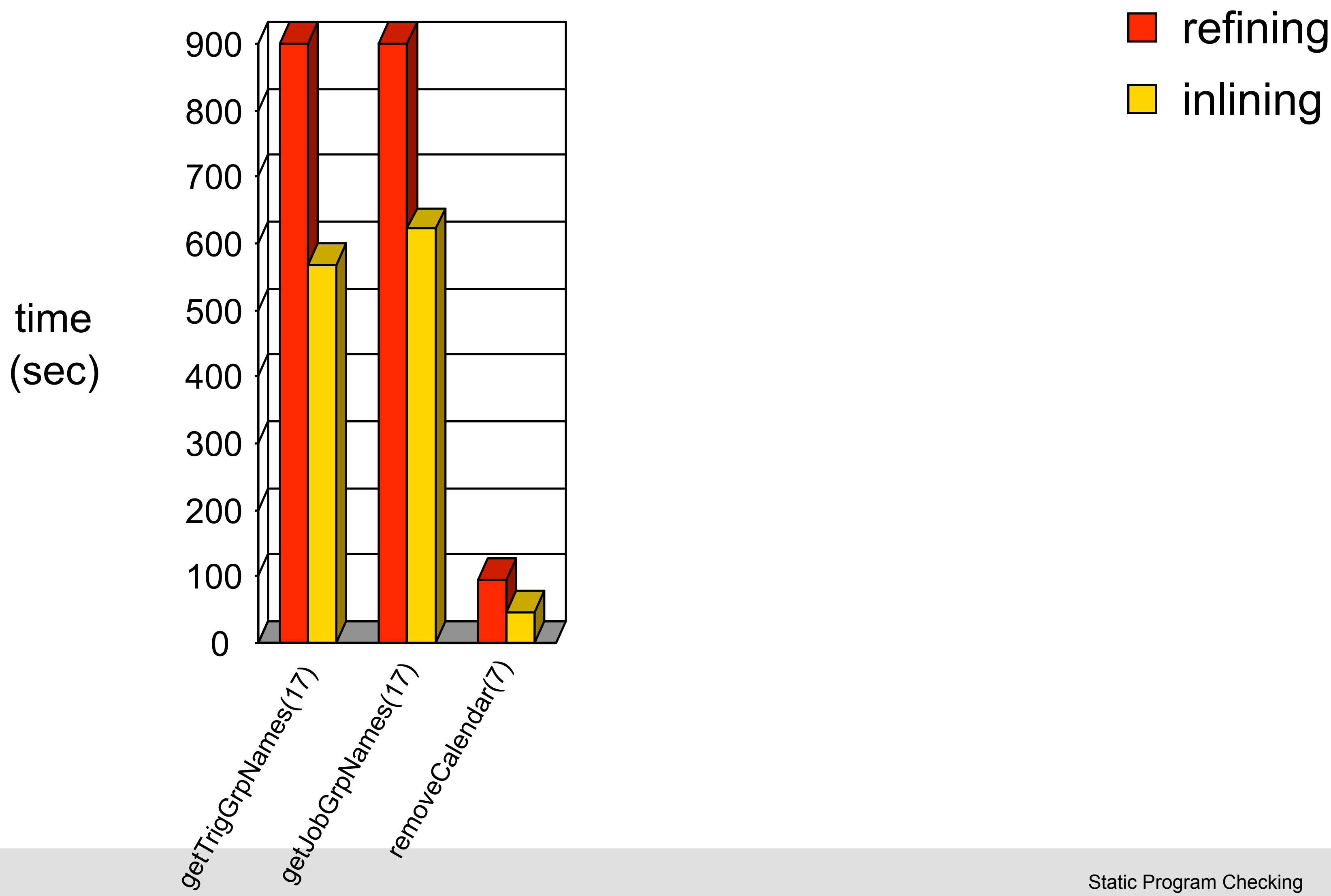
Component/s:	Triggers
Affects Version/s:	1.6
Fix Version/s:	1.6.1

File Attachments: Manage Attachments	1. QUARTZ-557-2.patch (1 kb) 2. QUARTZ-557.patch (0.8 kb)

Refining vs. inlining (partial properties)



Refining vs. inlining (full properties)



Abstract interp'n vs. frame cond'n vs. inlining