Static Program Checking

CEGAR-based Specification Inference

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
  
  \[
  \text{relational expr} \subseteq \text{field'/variable'} \subseteq \text{relational expr}
  \]
- The result can sometimes be precise
  
  \[
  \text{field'/variable'} = \text{relational expr}
  \]
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
- \text{Summary}(\text{proc}) \Rightarrow \text{spec}(\text{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

property

M

M1  M2

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 $M_1$ satisfies $\text{spec1}$ and $M_2$ satisfies $\text{spec2}$)
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.

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

```
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)
```

- map
- reachability
- quantifiers
- set cardinality
- list
Insight

Inferring complete specs is infeasible in general
But, necessary specs need not be complete

property

M

spec1
M1

spec2
M2
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

```java
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;
}
```

$l = [ j0, j1, j2, j3 ]$
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

```java
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

```java
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:

- `l = [ j0, j1, j2, j3 ]`
- `visitedPredsNum = 1`
- `visitedPredsNum = 0`
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;
}
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;
}
Eventually schedules all jobs

(l.jobs’ = l.jobs)
Example – call site specs

Property can be checked with full specs for all call sites

(l.jobs’ = l.jobs)

```java
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

\[(l.jobs' = l.jobs)\]

Property can be checked with full specs for all call sites

But, for some calls, \textit{partial specs} are enough

```java
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 – call site specs

Property can be checked with full specs for all call sites
But, for some calls, partial specs are enough

```java
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.jobs’ = l.jobs`  
`$\text{ret} \subseteq \text{cur}.*\text{next}`  
`jobs’ = jobs`  
`jobs’ = jobs`  
`Swaps the jobs of the given entries`
Example – call site specs

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

- program
- abstract
- spec
- refine spec
- invalid call
- check validity
- valid?
- invalid?
- instance
- solve
- negated property
- unsat?
- sat?
- no counter example
- counter example: bug!
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 → ℙInstance
solve(\( f, i \) = \{ i' | (i \subseteq i') \land (i' \in [f]) \})
- Solves a formula with respect to a partial instance

invalidate: Formula × Instance → Formula
(invalidate(\( f, i \) = g) \Rightarrow (solve(g, i) = \emptyset) \land (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

\[
\text{procedure } p() \{ \\
\quad \text{stmt1; } \underline{\text{q(); spec(q)}} \; \underline{\text{stmt2;}} \\
\}
\]

- 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

```java
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

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;
  }
}

\[
\begin{array}{l}
\text{return } = ?_{\text{Entry}} \\
\text{job'} = ? \\
\text{next'} = ? \\
\text{head'} = ? \\
\text{predecessors'} = ? \\
\ldots \\
\end{array}
\]

\[
\begin{array}{l}
\text{visitedPredsNum'} = ?_{\text{Job} \rightarrow \text{Int}} \\
\text{job'} = ? \\
\text{next'} = ? \\
\text{head'} = ? \\
\ldots \\
\end{array}
\]
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;
}

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

return = ?Entry
job’ = job
next’ = next
head’ = head
predecessors’ = predecessors
...
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;
}
### Framework: validity check

**procedure** `p()` {
stmt1; \<S0> \ 
`q(); spec(q);` \<S1> \ 
stmt2; \<S2> \ 
}

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

**Diagram:**
- **Program** → **Logical formula** → **Solve** → **Negated property**
- **Instance**
- **Refine spec** → **Invalid call** → **Check validity**
- **Counter example: bug!**
- **Unsat?** → **Sat?** → **Valid?** → **No counter example**

---

*Static Program Checking*
Framework: spec refinement

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

\[ \text{preState}(q) \land \text{translate}(q) \land \text{postState}(q) = \text{false} \]

- New spec must eliminate \((\text{preState}(q), \text{postState}(q))\)
- \text{translate}(q) can be the new spec
- Too big!
Framework: spec refinement

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

\[ \text{preState}(q) \land \text{translate}(q) \land \text{postState}(q) = \text{false} \]
- **Proof of unsatisfiability:**
  An unsatisfiable consequence of the solved formula
  \[ \text{translate}(q) \Rightarrow \text{proof} \]
  \[ \text{preState}(q) \land \text{proof} \land \text{postState}(q) = \text{false} \]
Example – spec refinement

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;
}

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;
}

Translation of `findReady`
(one unrolling)

Pre-state

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

Post-state

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

(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;
}

Translation of findReady (one unrolling)

Pre-state
(e = E0) (E0.job = J0) (E0.next = null)
...
($ret = E1) (E1.job = J1) (E1.next = null)
...

Post-state
(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)

false
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;` }

Translation of `findReady` (one unrolling)

<table>
<thead>
<tr>
<th>Pre-state</th>
<th>Post-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e = E0)</td>
<td>($ret = E1)</td>
</tr>
<tr>
<td>(E0.job = J0)</td>
<td>(E1.job = J1)</td>
</tr>
<tr>
<td>(E0.next = null)</td>
<td>(E1.next = null)</td>
</tr>
</tbody>
</table>

`Translation of findReady (one unrolling)`

<table>
<thead>
<tr>
<th>Translation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(e = null)</td>
<td>($ret = e)</td>
</tr>
<tr>
<td>((e != null) ∧ (e.job.predsNum = e.job.visitedPredsNum))</td>
<td>($ret = e)</td>
</tr>
<tr>
<td>((e != null) ∧ (e.job.predsNum != e.job.visitedPredsNum))</td>
<td>($ret = e.next)</td>
</tr>
<tr>
<td>(e.next = null)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New spec:</th>
</tr>
</thead>
<tbody>
<tr>
<td>($ret = e)</td>
</tr>
</tbody>
</table>
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

invalidate(q, s, s’)

empty spec  frame conditions  spec(q)  needed spec  full spec

Complexity of spec increases

Other ideas for ‘invalidate’?
- Translate(q) and S and S’ = False
Spec refinement: pace of progress

pre-state (s)

procedure q(..) {
  .
  .
  .
}

post-state (s')

case-based:
\neg (s \land 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
- case-based

empty spec  frame cond’n initial spec  needed spec  full spec

pre-state (s)
- $c_1$
- $a.g' = z_1$
- $x' = x_1$
- $y' = y_1$
- $x'.f' = y'$

post-state ($s'$)
- $c_2$
- $x' = x_2$
- $y' = y_2$
- $a.g' = \text{foo}()$

exec path:

$(\neg c_1 \land c_2) \Rightarrow f' = f + x_2 \rightarrow y_1 \land g' = g + a \rightarrow \text{foo} \_\text{ret}$

• pros / cons?
Spec refinement: pace of progress

- exec path
- case-based

empty spec  frame cond’n initial spec  needed spec  full spec

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

pre-state (s)

c1

a.g’ = z1
x’ = x1
y’ = y1
x’.f’ = y’

T

F

a.g’ = foo()
x’ = x2
y’ = y2

exec path:
(¬c1 ∧ c2) ⇒ f’ = f ++ x2 → y1 ∧
g’ = g ++ a → foo_ret

post-state (s’)

T

F

T

F
Spec refinement: pace of progress

- unsat proof
- exec path
- case-based

empty spec  frame cond’n initial spec  needed spec  full spec

pre-state (s)

- c1
- a.g’ = z1
- x’ = x1
- y’ = y1
- x’.f’ = y’

post-state (s’)

- c2
- a.g’ = foo()
- x’ = x2
- y’ = y2
- x’.f’ = y’

unsat proof:

c2 ⇒ f’ = (f ++ x1 → y1) ||
f’ = (f ++ x2 → 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 $\subseteq$ field'/variable' $\subseteq$ 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;
}

\[
\text{$ret \subseteq (e.*next \& (null + job.predsNum.(e.*next.job.visitedPredsNum)))$}
\]

\[
\text{$ret \supseteq \emptyset$}
\]
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
1\textsuperscript{st} 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
Refining vs. inlining (partial properties)
Refining vs. inlining (full properties)

- Refining
- Inlining

<table>
<thead>
<tr>
<th>Function</th>
<th>Refining (sec)</th>
<th>Inlining (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTrigGrpNames(17)</td>
<td>890</td>
<td>60</td>
</tr>
<tr>
<td>getUobGrpNames(17)</td>
<td>700</td>
<td>600</td>
</tr>
<tr>
<td>removeCalendar(17)</td>
<td>400</td>
<td>400</td>
</tr>
</tbody>
</table>

- Time (sec)
Abstract interp’n vs. frame cond’n vs. inlining

![Bar chart comparing abstract interpretation, frame condition, and inlining times for various functions.](chart_image)