Second-Order Constraints in Dynamic Invariant Inference

Kaituo Li (University of Massachusetts Amherst)
Christoph Reichenbach (Goethe University Frankfurt)
Yannis Smaragdakis (University of Athens)
Michal Young (University of Oregon)

21 August 2013
Invariants on Program Behaviour

- Invariants are formal documentation:
  - Pre-/postconditions, class invariants
  - Effect specifications
  - Dependency specifications
  
  Example:
  - ArrayStack.peek()
    requires
    preconditions:
    storageArray != null
    topOfStackIndex >= 0
    topOfStackIndex < storageArray.length
    ...

- Formal semantics: amenable to formal methods, testing
Invariants on Program Behaviour

- Invariants are formal documentation:
  - Pre-/postconditions, class invariants
  - Effect specifications
  - Dependency specifications
    
- Formal semantics: amenable to formal methods, testing

Example:

```java
ArrayStack.peek() requires preconditions:
  storageArray != null
  topOfStackIndex >= 0
  topOfStackIndex < storageArray.length
```

...
Limitations of current invariant-based approaches

- Invariants sometimes hard to write
  *Partly addressed by invariant inference*  
  (e.g., Daikon, DIDUCE, DySy, Heureka, ...)  
- Can bury important information  
  E.g., some Daikon-inferred specifications for simple methods have dozens of axioms  
- Can involve redundancy  
- Can be inconsistent with higher-level knowledge

ArrayStack.peek() requires:
  array != null  
  topOfStack >= 0  
  topOfStack < array.length

ArrayStack.pop() requires:
  array != null  
  topOfStack >= 0  
  topOfStack < array.length-1
Limitations of current invariant-based approaches

- Invariants sometimes hard to write
  *Partly addressed by invariant inference*
  (e.g., Daikon, DIDUCE, DySy, Heureka, . . .)
- Can bury important information
  E.g., some Daikon-inferred specifications for simple methods have dozens of axioms
- Can involve redundancy
- Can be inconsistent with higher-level knowledge

```
ArrayStack.peek() requires:
array != null
topOfStack >= 0
topOfStack < array.length
...

ArrayStack.pop() requires:
array != null
topOfStack >= 0
topOfStack < array.length - 1
...
```
Our approach: *constraints between sets of invariants*
Second-Order Constraints

Our approach: constraints between sets of invariants

Examples:

- if we meet all preconditions of ‘peek’ we also meet all preconditions of ‘pop’:
  \texttt{\textsc{Subdomain}(peek, pop)}
Our approach: *constraints between sets of invariants*

Examples:
- if we meet all preconditions of ‘peek’ we also meet all preconditions of ‘pop’:
  \( \text{SUBDOMAIN}(\text{peek, pop}) \)

ArrayStack.peek() requires: \( \text{SUBDOMAIN}(\text{peek, pop}) \)
  - array \(!=\) null
  - topOfStack \(\geq\) 0
  - topOfStack \(<\) array.length

...
Second-Order Constraints

Our approach: *constraints between sets of invariants*

Examples:

- if we meet all preconditions of ‘peek’ we also meet all preconditions of ‘pop’:
  \[\text{SUBDOMAIN}(\text{peek}, \text{pop})\]

- ‘pop’ is safe to call after ‘push’:
  \[\text{CANFOLLOW}(\text{push}, \text{pop})\]
Our approach: *constraints between sets of invariants*

Examples:

- if we meet all preconditions of ‘peek’ we also meet all preconditions of ‘pop’:
  \[ \text{SUBDOMAIN}(\text{peek, pop}) \]

- ‘pop’ is safe to call after ‘push’:
  \[ \text{CANFOLLOW}(\text{push, pop}) \]

- ‘\text{invert3x3Matrix}’ works like ‘\text{invertMatrix}’, but may have stronger preconditions:
  \[ \text{CONCORD}(\text{invertMatrix, invert3x3Matrix}) \]
Applying second-order constraints to Daikon

- **Program**
  - **Functionality**
  - **Test Suite**
  - **Observe**
  - **Instrument**

**Invariant Inference** (Daikon)

**Generate**

1st-order invariants
Applying second-order constraints to Daikon

- Hand-written documentation
Applying second-order constraints to Daikon

- Hand-written documentation
- Two software tools:
  - A Hints for first-order invariant inference

Diagram:
- Program
  - Functionality
  - Test Suite
  - Observe
  - Instrument
  - Invariant Inference (Daikon)
  - Inform
  - Generate
    - 2nd-order constraints
    - 1st-order invariants
Applying second-order constraints to Daikon

- Hand-written documentation
- Two software tools:
  A Hints for first-order invariant inference
  B *Automatically inferred* higher-level documentation

Program

Functionality

Test Suite

Invariant Inference (Daikon)

Inform

2nd-order constraints

Invariant Abstraction

1st-order invariants

Observe

Instrument

Generate
A catalogue of second-order constraints

- **SUBDOMAIN** *(peek, pop)*
  ‘pop’ is applicable whenever ‘peek’ is
A catalogue of second-order constraints

- **SUBDOMAIN** (peek, pop)
  - ‘pop’ is applicable whenever ‘peek’ is

- **SUBRANGE** (intersect, clear)
  - ‘clear’ ensures at least as many invariants as ‘intersect’
A catalogue of second-order constraints

- **Subdomain** (peek, pop)
  - ‘pop’ is applicable whenever ‘peek’ is

- **Subrange** (intersect, clear)
  - ‘clear’ ensures at least as many invariants as ‘intersect’

- **CanFollow** (connect, send)
  - ‘connect’ provides the requirements for ‘send’ is
A catalogue of second-order constraints

- **Subdomain (peek, pop)**
  ‘pop’ is applicable whenever ‘peek’ is

- **Subrange (intersect, clear)**
  ‘clear’ ensures at least as many invariants as ‘intersect’

- **CanFollow (connect, send)**
  ‘connect’ provides the requirements for ‘send’ is

- **Follows (push, pop)**
  Anyone calling ‘pop’ may call ‘push’ first
A catalogue of second-order constraints

- **SUBDOMAIN**(peek, pop)
  ‘pop’ is applicable whenever ‘peek’ is

- **SUBRANGE**(intersect, clear)
  ‘clear’ ensures at least as many invariants as ‘intersect’

- **CANFOLLOW**(connect, send)
  ‘connect’ provides the requirements for ‘send’ is

- **FOLLOWS**(push, pop)
  Anyone calling ‘pop’ may call ‘push’ first

- **CONCORD**(computeGeneralFFT, computeCoprimeFFT)
  ‘computeCoprimeFFT’ has behavioural subtype of ‘computeGeneralFFT’
A catalogue of second-order constraints

- **Subdomain** (peek, pop)
  ‘pop’ is applicable whenever ‘peek’ is

- **Subrange** (intersect, clear)
  ‘clear’ ensures at least as many invariants as ‘intersect’

- **CanFollow** (connect, send)
  ‘connect’ provides the requirements for ‘send’ is

- **Follows** (push, pop)
  Anyone calling ‘pop’ may call ‘push’ first

- **Concord** (computeGeneralFFT, computeCoprimeFFT)
  ‘computeCoprimeFFT’ has behavioural subtype of ‘computeGeneralFFT’

Other useful properties are conceivable
Logical structure of our constraint catalogue

<table>
<thead>
<tr>
<th>requires</th>
<th>ensures</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre (A)</td>
<td>post (A)</td>
</tr>
<tr>
<td>pre (B)</td>
<td>post (B)</td>
</tr>
</tbody>
</table>
Logical structure of our constraint catalogue

\[\text{requires} \quad \begin{align*}
\text{pre (A)} \\
\text{Subdomain} \\
\text{pre (B)}
\end{align*} \quad \text{ensures} \quad \begin{align*}
\text{post (A)} \\
\text{post (B)}
\end{align*}\]
Logical structure of our constraint catalogue

**requires**

pre (A)

\[ \text{Subdomain} \]

pre (B)

**ensures**

post (A)

\[ \text{Subrange} \]

post (B)
Logical structure of our constraint catalogue

- **requires**
  - \( \text{pre (A)} \)
  - \( \text{pre (B)} \)

- **ensures**
  - \( \text{post (A)} \)
  - \( \text{post (B)} \)

Relationships:
- **FOLLOWS**
- **SUBDOMAIN**
- **CanFollow**
- **SUBRANGE**
Logical structure of our constraint catalogue

requires

pre (A) \(\wedge\) Subdomain \(\Rightarrow\) CanFollow \(\Rightarrow\) post (B)

post (B) \(\Rightarrow\) Subrange \(\Rightarrow\) pre (B)

ensures

post (A) \(\Leftarrow\) Concord \(\Rightarrow\) pre (A)

pre (A) \(\Leftarrow\) post (A)
A  Refined First-Order Inference

- User provides hand-written second-order constraints

B  Second-Order Inference

- System infers second-order constraints
Refined First-Order Inference

```java
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";
```

ArrayStack.peek() `requires` ArrayStack.pop() `requires`
ArrayStack peek() requires ArrayStack.pop() requires
ArrayStack $s = \textbf{new} \text{ArrayStack}();$

$⇒ s.\text{push("") bar");}$

$⇒ s.\text{push("") foo");}$

$\textbf{assert} \ s.\text{peek()} == "foo"; \newline$

$\textbf{assert} \ s.\text{pop()} == "foo"; \newline$

$\textbf{assert} \ s.\text{peek()} == "bar"; \newline$

ArrayStack.peek() \textbf{requires} \quad \text{ArrayStack.pop() requires} \quad \textbf{requires}
Refined First-Order Inference

ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";

ArrayStack.peek() requires ArrayStack.pop() requires
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
⇒ assert s.peek() == "foo";
⇒ assert s.pop() == "foo";
⇒ assert s.peek() == "bar";

ArrayStack.peek() requires array != null
tos = 1

dtos = 1
tos >= 0
Refined First-Order Inference

```
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";
```

ArrayStack.peek() requires
array != null
tos = 1

ArrayStack.pop() requires
array != null
tos = 1
Refined First-Order Inference

ArrayStack \( s = \textbf{new} \) ArrayStack();
\( s . \text{push("bar")}; \)
\( s . \text{push("foo")}; \)
\textbf{assert} \( s . \text{peek()} == \text{"foo"}; \)
\textbf{assert} \( s . \text{pop()} == \text{"foo"}; \)
⇒ \textbf{assert} \( s . \text{peek()} == \text{"bar"}; \)

ArrayStack.peek() \textbf{requires}
array != null
tos = 1...

ArrayStack.pop() \textbf{requires}
array != null
tos = 1
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
⇒ assert s.peek() == "bar";

ArrayStack.peek() requires
array != null
tos = 1
tos >= 0

ArrayStack.pop() requires
array != null
tos = 1
Refined First-Order Inference

ArrayStack \( s = \text{new} \) ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
⇒ assert s.peek() == "bar";

ArrayStack.peek() requires
array != null
tos = 1
tos >= 0

ArrayStack.pop() requires
array != null
tos = 1
ArrayStack $s = \textbf{new} \ \text{ArrayStack}();$
\begin{itemize}
  \item $s.$push("bar");
  \item $s.$push("foo");
  \item assert $s.$peek() == "foo";
  \item assert $s.$pop() == "foo";
  \item assert $s.$peek() == "bar";
\end{itemize}
**Refined First-Order Inference**

```java
ArrayStack s = new ArrayStack();
s.push("bar");
s.push("foo");
assert s.peek() == "foo";
assert s.pop() == "foo";
assert s.peek() == "bar";
```

### ArrayStack

<table>
<thead>
<tr>
<th>array</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;foo&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;bar&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

### Each second-order constraint propagates information across Daikon

ArrayStack.peek() requires ArrayStack.pop() requires
Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a \textit{confidence} metric on 2nd-order constraint

Example hypothesis: $\text{SUBRANGE}(\text{ArrayStack.pop, ArrayStack.peek})$

\begin{align*}
\text{peek()} & \textbf{ensures} \quad \text{return.class == String} \\
& \text{this has only one value} \\
\text{pop()} & \textbf{ensures} \quad \text{return.class == String} \\
& \text{this has only one value} \\
& \text{return has only one value}
\end{align*}
Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint

Example hypothesis: $\text{SUBRANGE}(\text{ArrayStack.pop}, \text{ArrayStack.peek})$

```
peek() ensures
  return.class == String
  this has only one value

pop() ensures
  return.class == String
  this has only one value
  return has only one value

post(peek) ⇒ return.class == String?

confidence = \frac{1}{1}
```
Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a confidence metric on 2nd-order constraint

Example hypothesis: \texttt{SUBRANGE(ArrayStack.pop, ArrayStack.peek)}

\begin{align*}
\text{peek()} & \textbf{ensures} \quad \text{return.class == String} \\
& \quad \text{this has only one value} \\
\text{pop()} & \textbf{ensures} \quad \text{return.class == String} \\
& \quad \text{this has only one value} \\
& \quad \text{return has only one value}
\end{align*}

\[
\text{post(peek)} \Rightarrow \text{this has only one value?}
\]

\[
\text{confidence} = \frac{1+1}{1+1}
\]
Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint

Example hypothesis: $\text{SUBRANGE}(\text{ArrayStack.pop, ArrayStack.peek})$

```plaintext
peek()  \textbf{ensures} \\
return.class == String \\
this has only one value

pop()  \textbf{ensures} \\
return.class == String \\
this has only one value \\
return has only one value

post(peek) \Rightarrow return has only one value? \\

\text{confidence} = \frac{1+1+0}{1+1+1}
```
Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint
- Incorporate per-invariant **Daikon Confidence** metrics

Example hypothesis: **SUBRANGE**(ArrayStack.pop, ArrayStack.peek)

<table>
<thead>
<tr>
<th>DC</th>
<th>peek() ensures</th>
<th>DC</th>
<th>pop() ensures</th>
</tr>
</thead>
<tbody>
<tr>
<td>.9</td>
<td>return.class == String</td>
<td>.7</td>
<td>return.class == String</td>
</tr>
<tr>
<td>.8</td>
<td><em>this</em> has only one value</td>
<td>.6</td>
<td><em>this</em> has only one value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.5</td>
<td><em>return</em> has only one value</td>
</tr>
</tbody>
</table>

\[
\text{confidence} = .9 \times .8 ( .7 + .6 + .5 )
\]
Hypothesise all possible 2nd-order constraints per class
Compute fraction of overlap (using automated theorem prover): determine a \textit{confidence} metric on 2nd-order constraint
Incorporate per-invariant \textbf{Daikon Confidence} metrics

Example hypothesis: \texttt{SUBRANGE(ArrayStack.pop, ArrayStack.peek)}

\textbf{DC} \ \texttt{peek()} \ \textbf{ensures} \ \begin{align*}
.9 & \text{ return.class == String} \\
.8 & \text{ this has only one value}
\end{align*}

\textbf{DC} \ \texttt{pop()} \ \textbf{ensures} \ \begin{align*}
.7 & \text{ return.class == String} \\
.6 & \text{ this has only one value} \\
.5 & \text{ return has only one value}
\end{align*}

\textit{confidence} = \frac{.9 \times .8}{(______)}
Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a confidence metric on 2nd-order constraint
- Incorporate per-invariant Daikon Confidence metrics

Example hypothesis: \texttt{SUBRANGE(ArrayStack.pop, ArrayStack.peek)}

\begin{align*}
\text{DC } \texttt{peek()} & \quad \textbf{ensures} \\
.9 & \quad \text{return.class == String} \\
.8 & \quad \text{this has only one value} \\

\text{DC } \texttt{pop()} & \quad \textbf{ensures} \\
.7 & \quad \text{return.class == String} \\
.6 & \quad \text{this has only one value} \\
.5 & \quad \text{return has only one value}
\end{align*}

\[
\text{post(peek) } \Rightarrow \text{return.class == String?}
\]

\[
\text{confidence } = \frac{.9 \times .8 (.7 \max)}{1}
\]
Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a confidence metric on 2nd-order constraint
- Incorporate per-invariant **Daikon Confidence** metrics

Example hypothesis: `SUBRANGE(ArrayStack.pop, ArrayStack.peek)`

```
DC  peek() ensures
.9  return.class == String
.8  this has only one value

DC  pop() ensures
.7  return.class == String
.6  this has only one value
.5  return has only one value
```

\[
\text{post}(\text{peek}) \Rightarrow \text{this has only one value?} \quad \Rightarrow \quad \text{confidence} = \frac{.9 \times .8 (\frac{.7 + .6}{1+1})}{1+1}
\]
Second-Order Inference

- Hypothesise all possible 2nd-order constraints per class
- Compute fraction of overlap (using automated theorem prover): determine a *confidence* metric on 2nd-order constraint
- Incorporate per-invariant **Daikon Confidence** metrics

Example hypothesis: **SUBRANGE**(ArrayStack.pop, ArrayStack.peek)

DC peek() ensures
  .9 return.class == String
  .8 *this* has only one value

DC pop() ensures
  .7 return.class == String
  .6 *this* has only one value
  .5 return has only one value

post(peek) ⇒ return has only one value?

\[
\text{confidence} = \frac{.9 \times .8 (.7 + .6 + 0)}{1 + 1 + 1}
\]
Evaluation (overview)

A  Refined First-Order Inference

- Evaluated changes to inferred first-order invariants
- Second-order constraints written by hand for:
  - Daikon’s StackAr
  - 18 Apache Commons Collections classes
  - 7 AspectJ classes

B  Second-Order Inference

- Evaluated generated second-order constraints
- For all hand-written second-order constraints
- For random classes:
  - 2 Apache Commons Collections
  - 2 AspectJ
- Confidence threshold: 0.75
Refined First-Order Inference

- Evaluated changes to inferred first-order invariants
- Second-order constraints written by hand for:
  - Daikon’s StackAr
  - 18 Apache Commons Collections classes
  - 7 AspectJ classes
StackAr: Daikon’s stack-based array implementation

- **int `topOfStack`:** top-of-stack location
- **int[] `theArray`:** stack representation
- **`pop`:** remove top element
- **`top`:** peek at top element value
- **`topAndPop`:** remove top element and return it
Evaluation: Refined Invariant Inference (1)

StackAr: Daikon’s stack-based array implementation

- `int topOfStack`: top-of-stack location
- `int[] theArray`: stack representation
- `pop()`: remove top element
- `top()`: peek at top element value
- `topAndPop()`: remove top element and return it

```
SUBDOMAIN(pop, top)
SUBDOMAIN(top, topAndPop)
SUBDOMAIN(topAndPop, pop)
```
StackAr: Daikon’s stack-based array implementation

- int topOfStack: top-of-stack location
- int[] theArray: stack representation
- pop(): remove top element
- top(): peek at top element value
- topAndPop(): remove top element and return it

\[
\begin{align*}
\text{Subdomain}(\text{pop, top}) \\
\text{Subdomain}(\text{top, topAndPop}) \\
\text{Subdomain}(\text{topAndPop, pop})
\end{align*}
\]

- Removed 5 false invariants:
  - this has only one value
  - theArray has only one value
  - theArray.length == 100
  - theArray[topOfStack] != null
  - topOfStack < theArray.length-1
StackAr: Daikon’s stack-based array implementation

- **int** topOfStack: top-of-stack location
- **int[]** theArray: stack representation
- **pop()**: remove top element
- **top()**: peek at top element value
- **topAndPop()**: remove top element and return it

Removed 5 false invariants:

Added 2 new invariants:

- `this.topOfStack >= -1`
- `DEFAULT_CAPACITY != theArray.length - 1`
StackAr: Daikon’s stack-based array implementation
  - int topOfStack: top-of-stack location
  - int[] theArray: stack representation
  - pop(): remove top element
  - top(): peek at top element value
  - topAndPop(): remove top element and return it

Removed 5 false invariants:
Added 2 new invariants:
Refined 1 overly-specific invariants:

```
this.topOfStack < size(this.theArray[])-1
⇓
this.topOfStack <= size(this.theArray[])-1
```
### Evaluation: Refined Invariant Inference (2)

<table>
<thead>
<tr>
<th>Experiment</th>
<th>2nd-order Constraints Written</th>
<th>1st-order Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackAr #1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>StackAr #2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Apache Commons Collections</td>
<td>26</td>
<td>25 + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>AspectJ</td>
<td>26</td>
<td>12 + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

- Most changes **positive**
- Only two **incorrect** invariants introduced
  - Both due to nonmonotonicity in the underlying first-order invariant inference mechanism
Evaluation: Refined Invariant Inference (2)

<table>
<thead>
<tr>
<th>Experiment</th>
<th>2nd-order Constraints Written</th>
<th>Added</th>
<th>1st-order Invariants Removed</th>
<th>Refined</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackAr #1</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>StackAr #2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Apache Commons Collections</td>
<td>26</td>
<td>25 + 1</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>AspectJ</td>
<td>26</td>
<td>12 + 1</td>
<td>12</td>
<td>3</td>
</tr>
</tbody>
</table>

- Most changes positive
- Only two incorrect invariants introduced
  - Both due to nonmonotonicity in the underlying first-order invariant inference mechanism

Overwhelmingly positive effects
Evaluation (overview)

B Second-Order Inference

- Evaluated generated second-order constraints
- For all hand-written second-order constraints
- For random classes:
  - 2 Apache Commons Collections
  - 2 AspectJ
- Confidence threshold: 0.75
Confirming our manual annotations:
Of our 64 original manual annotations:
- 37 we inferred
- 27 we did *not* infer, of which:
  - 12 we had wrongly annotated
  - 7 lacked any supporting data samples
  - 6 we rejected due to ‘noise’ first-order invariants
  - 2 were **Concord**
Confirming our manual annotations:
Of our 64 original manual annotations:
  - 37 we inferred
  - 27 we did \textit{not} infer, of which:
    - 12 we had wrongly annotated
    - 7 lacked any supporting data samples
    - 6 we rejected due to ‘noise’ first-order invariants
    - 2 were \textbf{Concord}

Finding new second-order constraints:

<table>
<thead>
<tr>
<th>Class</th>
<th>inferred</th>
<th>incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Commons: AbstractMapBag</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Apache Commons: SingletonMap</td>
<td>806</td>
<td>0</td>
</tr>
<tr>
<td>AspectJ Reflection:</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>AspectJ LstBuildConfigManager:</td>
<td>112</td>
<td>5</td>
</tr>
</tbody>
</table>
Confirming our manual annotations:
Of our 64 original manual annotations:
- 37 we inferred
- 27 we did not infer, of which:
  - 12 we had wrongly annotated
  - 7 lacked any supporting data samples
  - 6 we rejected due to ‘noise’ first-order invariants
  - 2 were Concord

Finding new second-order constraints:

<table>
<thead>
<tr>
<th>Class</th>
<th>Score</th>
<th>Num.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Commons: AbstractMapBag</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>Apache Commons: SingletonMap</strong></td>
<td>806</td>
<td>0</td>
</tr>
<tr>
<td>AspectJ Reflection:</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>AspectJ LstBuildConfigManager:</td>
<td>112</td>
<td>5</td>
</tr>
</tbody>
</table>

**Immutable** class: suggests even higher-order invariants!
Confirming our manual annotations:
Of our 64 original manual annotations:
- 37 we inferred
- 27 we did not infer, of which:
  - 12 we had wrongly annotated
  - 7 lacked any supporting data samples
  - 6 we rejected due to ‘noise’ first-order invariants
  - 2 were Concord

Finding new second-order constraints:

<table>
<thead>
<tr>
<th>Class</th>
<th>inferred</th>
<th>incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Commons: AbstractMapBag</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Incomplete unit tests</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AspectJ Reflection</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>AspectJ LstBuildConfigManager</td>
<td>112</td>
<td>5</td>
</tr>
</tbody>
</table>

⇒ Poor Daikon invariants
Conclusions

Second-order constraints:

- Permit high level of discourse about program properties
- Refine the quality of detected first-order invariants
- Can be detected automatically
- Are easy to use and powerful