Static Program Checking

Specification Inference

Jun.-prof. Dr. Mana Taghdiri

Monday – July 10, 2014
Statistical justification

- Only reports those invariants that are statistically justified
  - Not the ones that happen to be true by chance
- Depends on the set of values obtained at a program point
- Example 1:
  - In an entire test suite, a program point was executed only 3 times with \( x = 7, -42, 22 \)
  - The invariants \( x \neq 0, x \leq 22, x \geq -42 \) will be generated
- Example 2:
  - For \( 0 < y < 10, 0 < z < 10 \), given 3 pairs \( <y, z> \), the invariant \( y \neq z \) can be inferred.
  - Might be more reliable if true for 10000 pairs
- Solution?
Statistical justification

- Only reports those invariants that are statistically justified
  - Not the ones that happen to be true by chance
- Depends on the set of values obtained at a program point
- Example 1:
  - In an entire test suite, a program point was executed only 3 times with $x = 7, -42, 22$
  - The invariants $x \neq 0$, $x \leq 22$, $x \geq -42$ will be generated
- Example 2:
  - For $0 < y < 10$, $0 < z < 10$, given 3 pairs $<y, z>$, the invariant $y \neq z$ can be inferred.
  - Might be more reliable if true for 10000 pairs
- Solution 1:
  - Ask for a better test suite
    - But how to generate an ideal test suite?
Statistics justification

Daikon’s solution:

- For each detected invariant, it computes the probability that the invariant might appear by chance in a random input.
- If the probability is less than a user-provided threshold, then the property is not just by chance, and is reported.
- It assumes a distribution and performs a statistical analysis.
- Because the actual distribution of variable values is unknown, the computed probability is not absolute, but the exact value is not so important; the order of magnitude is important.
- Daikon uses uniform distribution of values, not a guarantee, but a good measure.
- User’s threshold must be very low.
  - Because Daikon checks for millions of invariants.
Statistical justification

- Daikon has a probability computation for each invariant
- Example
  - Suppose variable $x$ takes values in a range with size $r$ (based on our samples) containing 0
  - We have $s$ sample values of $x$
  - Suppose in all samples $x \neq 0$
  - Probability of this invariant?
**Statistical justification**

- Daikon has a probability computation for each invariant
- Example
  - Suppose variable $x$ takes values in a range with size $r$ (based on our samples) containing 0
  - Suppose in all samples $x \neq 0$
  - Assuming uniform distribution, probability of $x \neq 0$ in one sample $= 1 - 1/r$
  - Given $s$ sample values of $x$, probability $= (1 - 1/r)^s$
  - If this probability is less than the threshold, then report the invariant
- More precisely:
  - In an entire test suite, a point was executed only 3 times with $x = 7, -42, 22$
  - The invariants $x \neq 0$, $x \leq 22$, $x \geq -42$ will be generated
  - Probability of nonzero $= (1 - 1/65)^3 = 0.94$
  - So $x \neq 0$ will not be reported
Statistical justification

- The statistical heuristic is not a guarantee. So Daikon also outputs the number of values (samples) that support an invariant, so the user can decide

- Problematic case: repeated values
  - A variable is not changed in a loop, but recorded repeatedly at the loop entry
  - Then number of samples is artificially high
  - We don’t like properties derived based on that
Example revisited – add array elements

```
i, s := 0, 0;
do i ≠ n →
   i, s := i + 1, s + b[i]
od
```

Precondition: $n \geq 0$
Postcondition: $s = (\sum j : 0 \leq j < n : b[j])$
Loop invariant: $0 \leq i \leq n$ and $s = (\sum j : 0 \leq j < i : b[j])$

- Instrumentation at the program entry, the loop head, and program exit.
- Ran on 100 randomly-generated arrays of length 7-13 with elements from -100 to 100.
Example – add array elements

15.1.1:::ENTER
N = size(B)
N in [7..13]
B
All elements in [-100..100]

100 samples
(7 values)
(100 values)
(200 values)

15.1.1:::EXIT
N = I = orig(N) = size(B)
B = orig(B)
S = sum(B)
N in [7..13]
B
All elements in [-100..100]

100 samples
(7 values)
(100 values)
(96 values)
(7 values)
(100 values)
(200 values)

15.1.1:::LOOP
N = size(B)
S = sum(B[0..I-1])
N in [7..13]
I in [0..13]
I <= N
B
All elements in [-100..100]

1107 samples
(7 values)
(452 values)
(7 values)
(14 values)
(77 values)
(100 values)
(200 values)
(985 values)
(200 values)
Example – add array elements

- additional invariants if we don’t eliminate repeated values

- these invariants are reported for the loop, but nowhere else in the code, although B doesn’t change in the program

\[
\begin{align*}
\text{sum(B) in } [-556..539] & \quad \text{(96 values)} \quad \star \\
\text{B[0] nonzero in } [-99..96] & \quad \text{(79 values)} \quad \star \\
\text{B[-1] in } [-88..99] & \quad \text{(80 values)} \quad \star \\
N \neq B[-1] & \quad \text{(99 values)} \quad \star \\
B[0] \neq B[-1] & \quad \text{(100 values)} \quad \star
\end{align*}
\]
Repeated values – solutions

- **Always:**
  - Every sample should be counted, no matter what

- **Changed value:**
  - A sample is considered if its value is different from last time this program point was examined (doesn’t account for recomputations that result in the same value)

- **Assignment:**
  - Sample contributes to confidence computation if variable has been assigned since last time seen at this program point. (implemented in Daikon) – takes engineering effort to implement

- **Random:**
  - If value has changed, then consider it. Otherwise, consider it with a probability of \( \frac{1}{2} \)
Statistical confidence – bottom line

- An invariant is reported if:
  - There are enough samples that contribute to computing confidence (based on the picked strategy)
  - The computed confidence is higher than the user-specified threshold
Comparable variables – example

```c
// Return the sum of the elements of array b, which has length n.
long array_sum(int * b, long n) {
    long s = 0;
    for (int i=0; i<n; i++)
        s = s + b[i];
    return s;
}
```

Unconstrained: all scalars (array elements, indices, addresses)

Source type: (i, elements of b) (s, n)

Coerced: same as unconstrained

Lackwit: ?
Comparable variables – example

```c
// Return the sum of the elements of array b, which has length n.
long array_sum(int * b, long n) {
    long s = 0;
    for (int i=0; i<n; i++)
        s = s + b[i];
    return s;
}
```

Unconstrained: all scalars (array elements, indices, addresses)

Source type: (i, elements of b) (s, n)

Coerced: same as unconstrained

Lackwit: (i, n) (s, elements of b)
  ** if b[i] > 0, then i <= s, but this can’t be inferred by lackwit types
  ** doesn’t occur much in practice
Experimental results – comparable vars

<table>
<thead>
<tr>
<th>Comparability</th>
<th>Other vars</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconstrained</td>
<td>8.8</td>
<td>1.00</td>
</tr>
<tr>
<td>Source types</td>
<td>4.5</td>
<td>.51</td>
</tr>
<tr>
<td>Coerced types</td>
<td>5.1</td>
<td>.58</td>
</tr>
<tr>
<td>Lackwit</td>
<td>0.6</td>
<td>.06</td>
</tr>
</tbody>
</table>

Gives average number of variables to which each variable is comparable, and the ratios between each method and the unconstrained method.
Experimental results – effect on invariants

<table>
<thead>
<tr>
<th>Comparability</th>
<th>Invariants total</th>
<th>Invariants binary</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconstrained</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Source types</td>
<td>78%</td>
<td>61%</td>
<td>91%</td>
</tr>
<tr>
<td>Coerced types</td>
<td>78%</td>
<td>74%</td>
<td>96%</td>
</tr>
<tr>
<td>Lackwit</td>
<td>55%</td>
<td>27%</td>
<td>75%</td>
</tr>
</tbody>
</table>

Percentage of total and binary invariants reported, and time to compute all invariants, compared to unconstrained comparability

Qualitative analysis of invariants:
Those not reported by Lackwit are all irrelevant for common programming tasks
Example: other techniques produce $x < y$ (for char * pointers) – not useful
How to handle data structures?

- Sample invariants:

```
PriorityQueue::CLASS
   prio.closure(next).rank is sorted by <=

void PriorityQueue.insert()::EXIT
   size(prio.closure(next)) = size(orig(prio.closure(next))) + 1
```
Data structure invariants

- Pointers are difficult only for recursive data structures where the system may have to traverse arbitrarily many links
  - O.w. a pointer is just a record with two fields (address, and content)

- Invariants are
  - Local: true in objects with a fixed distance from the current variable
    - node.left.parent = node
    - The instrumenter records object fields up to a certain specified depth
  - Global: involves an arbitrary-size collection of objects
    - mytree is sorted
    - num < size(myList)
    - Must explicitly represent the collection
How to handle data structures?

- Linearization
  - Instrumented code traverses data structures and records them explicitly as arrays in program traces
  - Example invariant: mytree is sorted
linearization

- Linearization involves
  - Selecting a root
    - Current program variables
  - Determining a field for traversing the data structure
    - Fields that point to objects of the same type (next)
    - If there are multiple options (e.g. prev), then makes multiple arrays
    - Also for combinations of fields
      - (in-order, pre-order and post-order of left and right in a tree)
  - Selecting which fields of the visited objects should be written in trace file
    - Fields with non-recursive types are written out
  - Also records special attributes
    - Is cyclic, is a DAG, is a tree
    - This kind of information must be discovered by instrumenter. Is lost after linearization
Conditional invariants

- Many important invariants are not universal
  - p.left.value < p.right.value is true if p, p.right and p.left are non null
  - If arg < 0 then result = -arg else result = arg (absolute value)
  - If x \in orig(list) then size(list) = size(orig(list)) – 1 (list deletion)
Conditional invariants

- Many important invariants are not universal
  - p.left.value < p.right.value is true if p, p.right and p.left are non null
  - If arg < 0 then result = -arg else result = arg (absolute value)
  - If x \in orig(list) then size(list) = size(orig(list)) – 1 (list deletion)
Trace splitting

- What predicates to use for splitting?
- How many of the split candidates to use?
- How to combine them?
- Example
  - For two splitting predicates p and q, there are at least 13 potential subparts of data trace:
Trace splitting

- What predicates to use for splitting?
- How many of the split candidates to use?
- How to combine them?
- Example
  - For two splitting predicates \( p \) and \( q \), there are at least 13 potential subparts of data trace:
    - Whole trace (condition true)
    - Four subtraces \( (p, \neg p, q, \neg q) \)
    - Eight subtraces \( (p \land q, \neg(p \land q), p \land \neg q, \neg(p \land \neg q), \neg p \land q, \neg(p \land q), \neg p \land \neg q, \neg(p \land \neg q)) \)

- Daikon uses single-level splitting
  - True, and \( p, \neg p, q, \neg q \)
Splitting policy

- A static analysis policy:
  - splitting conditions based on analysis of the program’s source code
  - Daikon currently implements this policy
  - Uses conditions used for branches in program
    - (if statements and pure boolean member functions)

- A special values policy:
  - compares a variable to preselected values chosen
    - Statically (such as null, zero, or literals in the source code) or

- A policy based on exceptions to detect invariants:
  - tracks variable values that violate potential invariants, rather than immediately discarding the falsified invariant
  - If the number of falsifying samples is moderate, those samples can be separately processed, resulting in a nearly-true invariant plus an invariant over the exceptions

- A programmer-directed policy:
  - allows a user to select splitting conditions a priori
Experiments on data structures

Redundant (implied) invariant detection was not implemented for this experiment

<table>
<thead>
<tr>
<th>class</th>
<th>relevant</th>
<th>implied</th>
<th>irrelevant</th>
<th>precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>317</td>
<td>11</td>
<td>1</td>
<td>96%</td>
</tr>
<tr>
<td>OrderedList</td>
<td>201</td>
<td>5</td>
<td>5</td>
<td>95%</td>
</tr>
<tr>
<td>StackLi</td>
<td>184</td>
<td>8</td>
<td>1</td>
<td>95%</td>
</tr>
<tr>
<td>StackAr</td>
<td>159</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>QueueAr</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>ListNode</td>
<td>46</td>
<td>1</td>
<td>1</td>
<td>95%</td>
</tr>
<tr>
<td>LinkedListItr</td>
<td>185</td>
<td>8</td>
<td>0</td>
<td>95%</td>
</tr>
</tbody>
</table>
Experiments on data structures

<table>
<thead>
<tr>
<th>class</th>
<th>relevant</th>
<th>missing</th>
<th>recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>317</td>
<td>3</td>
<td>99%</td>
</tr>
<tr>
<td>OrderedList</td>
<td>201</td>
<td>5</td>
<td>98%</td>
</tr>
<tr>
<td>StackLi</td>
<td>184</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>StackAr</td>
<td>159</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>QueueAr</td>
<td>500</td>
<td>10</td>
<td>98%</td>
</tr>
<tr>
<td>ListNode</td>
<td>46</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>LinkedListItr</td>
<td>185</td>
<td>2</td>
<td>99%</td>
</tr>
</tbody>
</table>

These are all textbook data structures, so we know the exact set of invariants
Sample invariants

```java
void LinkedList.insert(Object x, LinkedListItr p)::EXIT
    x = header.next.element
    if (p != null && p.current != null)
        then size(orig(header.next.closure(next))) = size(header.next.closure(next)) - 1
        else header.close(next) = orig(header.close(next))

boolean LinkedList.isEmpty()::EXIT
    if (header.next == null)
        then return = true
        else return = false

void LinkedList.remove(Object x)::EXIT
    size(header.next.closure(next)) <= size(orig(header.next.closure(next)))
    MISSING: if (findPrevious(s) != null)
        then size(header.next.closure(next)) = size(orig(header.next.closure(next))) - 1
        else size(header.next.closure(next)) = size(orig(header.next.closure(next)))
```
Final words

- Experiments show the invariants are
  - Accurate, useful, and efficient to generate

- Scalability
  - Even relatively small test suites are enough for detecting good invariants

- Ease of use
  - Still uses a lot of memory (internal data structures)
  - Usability can be improved (to cope with all generated invariants)
Static summary computation

- Summaries specify how a procedure behaves
  - Computed in the *Alloy* language
  - Are safe (sound) abstractions
    - It is guaranteed that the summaries account for all procedure executions

- This summarization technique
  - Is fully automatic
  - Requires absolutely *no annotations/guidance/additional information from the user*

- Goal: *cost-effectiveness*
  - The summarization must be fast
    - Is used as a small phase of a bigger bug finding technique
    - Is *linear in the size of the code*
  - Accuracy is not so important
    - Produces as accurate summaries as possible using a lightweight technique
Syntactic summaries

- Summarize 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.
- Provide 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.
  
  \[
  \text{field'/variable'} = \text{relational expr}
  \]
Example – Precise Spec

Job nullifyMove(Entry e1, Entry e2) {
    e1.job = e2.job;
    e2.job = null;
    return e1.job;
}
Example – Precise Spec

Job nullifyMove(Entry e1, Entry e2) {
    e1.job = e2.job;
    e2.job = null;
    return e1.job;
}

The summary is correct even when e1 and e2 are aliased.

relational override

job' = job ++ (e1 \rightarrow e2.job) ++ (e2 \rightarrow null)
$ret = e1.(job ++ (e1 \rightarrow e2.job) ++ (e2 \rightarrow null))
Example – Imprecise Spec

In a list of jobs, returns the first one with n predecessors

Entry findFirst(Entry e, int n) {
    Entry c = e;
    while ((c != null) && (c.job.predsNum != n)) {
        c = c.next;
    }
    return c;
}

$\text{ret} \subseteq e.\text{next}$

$\text{ret} \supseteq \emptyset$
Example – Imprecise Spec

Entry findFirst(Entry e, int n) {
    Entry c = e;
    while ((c != null) && (c.job. predsNum != n)) {
        c = c.next;
    }
    return c;
}

\[ \text{Return value is reachable from } e \]
And it is either null, or its f field equals d

\[ \text{c.f } \neq \text{ d} \]

\[ \text{\$ret } \subseteq (e.\text{next } \& (\text{null } + \text{f.d})) \]

\[ \text{\$ret } \supseteq \emptyset \]
Example – Imprecise Spec

Entry findFirst(Entry e, int n) {
    Entry c = e;
    while ((c != null) && (c.job.predsNum != n)) {
        c = c.next;
    }
    return c;
}

Return value is reachable from e
And it is either null, or its job.predsNum field equals n

\[ \text{return } c; \]

Why is this imprecise?

\[ \text{Return value is reachable from } e \]
\[ \text{And it is either null, or its } \]
\[ \text{job.predsNum field equals } n \]

\[ \text{Return value is reachable from } e \]
\[ \text{And it is either null, or its } \]
\[ \text{job.predsNum field equals } n \]

\[ \text{Return value is reachable from } e \]
\[ \text{And it is either null, or its } \]
\[ \text{job.predsNum field equals } n \]

\[ \text{Return value is reachable from } e \]
\[ \text{And it is either null, or its } \]
\[ \text{job.predsNum field equals } n \]
Example – Imprecise Spec

```java
void JobList.init() {
    Entry c = this.head;
    while (c != null) {
        c.job.scheduled = 0;
        c = c.next;
    }
}
```

• The scheduled field of any job reachable from this list may be changed to 0

• The scheduled field of all other jobs remain unchanged

scheduled’ ⊆ scheduled + (this.head.*next.job → 0)
scheduled’ ⊇ scheduled - (this.head.*next.job → univ)
Approach: abstract interpretation

- Study aspects of the concrete (but more complicated) executions by looking at corresponding properties of abstract (and simpler) executions.

- Example: for the abstract domain of \{(+), (−), (+/−)\}
  - \(-1515 \times 17 \Rightarrow -(+) \times (+) \Rightarrow (−)\times (+) \Rightarrow (−)\)
  - \(-1515 + 17 \Rightarrow -(+) + (+) \Rightarrow (−) + (+) \Rightarrow (+/−)\)

- Abstract interpretation approximates program behavior by replacing the concrete domain of computation and its concrete operations with an abstract domain and abstract operations.
Value abstractions

\[ 2 \in \text{even-int} \quad \text{nonnegative} \]

\[ \{2\} \quad \{2, 5\} \quad \text{rational} \]

All the properties listed on the right are abstractions of 2; the upwards lines denote \(\subseteq\), a loss of precision.
Concretization function

```
concrete value sets

{ -1, 2 } <- { 2 } <- { f } <- { t }
{ 2, 5 } <- { t,f,2 } <- { 0..9 } <- int
{ 0,1,2,... } <- { 0,1,2,... } <- nonneg
{ ..., -1, 0, 1, ... } <- ... <- abstract values

AllData
```
Abstraction function

Function $\alpha$ maps each set to the abstract value that best describes it.
Abstract interpretation

- Consists of
  - A concrete domain $S$, and an abstract domain $A$
  - An abstraction function $\alpha$, and a concretization function $\gamma$
  - $\alpha$ and $\gamma$ form a Galois connection
    - $S \in \gamma(\alpha(S))$
    - $A = \alpha(\gamma(A))$

- Is defined over an ordered set (lattice)
  - a partially ordered set where any two elements have a unique least upper bound (join) and a greatest lower bound (meet)

- Examples:
  - The set $\{1, 2, 3\}$ and the subset relation
    - Bounded with a bottom and a top
  - The set of natural numbers and the less-than relation
    - Unbounded with a bottom
Technique: Abstract Interpretation

Abstract domain = <Lower bound, Upper bound>

\[ E^\ell, E^u : (\text{Var} \cup \text{Field} \cup \text{Type}) \rightarrow \text{Relational Expr} \]

Partial order

\[ <E^\ell_1, E^u_1> \sqsubseteq <E^\ell_2, E^u_2> \iff \forall x, (E^\ell_1(x) \supseteq E^\ell_2(x)) \land (E^u_1(x) \subseteq E^u_2(x)) \]

Lattice join

\[ <E^\ell_1, E^u_1> \sqcup <E^\ell_2, E^u_2> = \langle \lambda x. E^\ell_1(x) \& E^\ell_2(x), \lambda x. E^u_1(x) + E^u_2(x) \rangle \]
Approach

- Is an abstract interpretation
  - Flow sensitive (order of statements is important)
  - Context sensitive (calling context is important)
- Abstract domain
  - Relational expressions in Alloy
- Symbolic execution
  - Pre-state
    - Each type, variable, field is represented by a relation constant
  - Execution
    - Keeps two expressions for each type, variable, field:
      - Lower bound: tuples that occur in all executions (must side-effects)
      - Upper bound: tuples that occur in some executions (may side-effects)
    - As the code updates the values, the relational expressions are updated
  - Final summary is an over-approximation of the behavior

\[(lb \in x') \land (x' \in ub)\]
Example

1. x.f = y;
2. if (c == 1)
3.    z.g = y;
4. else
5.    z.g = x;
6. end

0. x, y, z, f, g, c
1. [f] = f ++ x -> y
3.  [g] = g ++ z -> y
5.  [g] = g ++ z -> x
6.  [g] ⊆ g + z -> y + z -> x
6.  [g] ⊇ g - z -> univ
Program constructs

- Object allocation
  - Allocated objects are represented by fresh unary relations

- Call sites
  - Context-sensitive summaries
  - Context consists of variables, fields, types whose summaries are accurate
  - Generate a template summary for each context of a procedure using symbolic constants for accessed fields, variables, types
  - Instantiate it at each call site using relational expressions of fields, variables, and types at that point

- Loops
  - Use the loops condition to get more precise abstractions of the body
    - At the entry point, the relational encoding of the condition is intersected with expressions for the condition’s variables to remove all tuples that violate the condition
    - Then abstract the body by computing fixpoint
    - Then intersect the negation of loop condition with the final values of condition’s variables
Technique: Abstract Interpretation

Widenings:

- $x + x.r + \ldots + x.r^{(k)}$ in upper bound goes to $x.*r$
  
- $x \& x.r \& \ldots \& x.r^{(k)}$ in lower bound goes to $\emptyset$

- upper bound with more than $n$ operators goes to $\text{univ}$
  
- lower bound with more than $n$ operators goes to $\emptyset$

- $(m+1)^{\text{th}}$ allocation of a type goes to a symbolic set of objects with unspecified cardinality

- simplification rules to shorten the exprs