Karlsruhe Institute of Technology

Static Program Checking

Invariant detection – Daikon

Automated Software Analysis Group, Institute of Theoretical Informatics

Jun.-prof. Dr. Mana Taghdiri

Monday – July 3, 2014

Invariants

Karlsruhe Institute of Technology

- What is an invariant?
  - A property that is true at a particular program point or points
  - Like the ones written as assert statements, rep invariants, pre/post conditions
- Having an explicit invariant simplifies
  - Coding
  - Verification, Testing
  - Optimization
  - Maintenance
  - Understanding data structures, algorithms, program operations
- All programmers have invariants in mind when coding
  - An idea of how the system in intended to be used
  - How the data structures are laid out
- But, invariants are usually absent from the code
  - Automatic invariant detection recovers what programmer had in mind

Automatic invariant detection

- Can be done statically
  - One approach is abstract interpretation (will see an example in the next class)

  - Houdini
    - Generates rep invariants, pre/post conditions
    - But not assert statements in the middle of the code
    - Generates all possible candidate invariants
    - Refutes the invalid ones by iteratively calling ESC/Java
    - The invariants are not guaranteed to be sound
    - But, they are true in all executions that ESC checks

Daikon

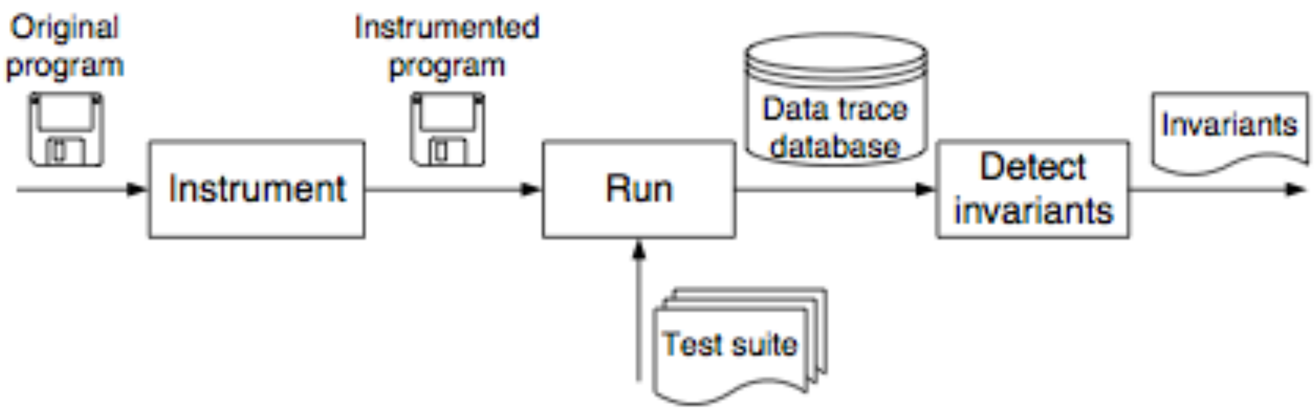
- Uses a dynamic approach
  - Based on a set of program traces
  - Executes a test suite
  - Captures variable values at program points of interest
  - An invariant detector determines which properties hold for variables
  - Runs very quickly on large programs

- The quality of the output depends on the comprehensiveness of the test suite
  - Daikon infers “likely” invariants
  - Experiments show that test suites found in practice are adequate

Quality of test suite

- Invariants generated by Daikon can be used to enhance the suite
  - The programmer sees the invariants that are true so far, but shouldn't hold in general, and can come up with other test cases
- Test suites that are good for finding bugs are not necessarily good for detecting invariants:
  - In bug finding, for efficiency, every statement is covered a minimal number of times
  - In invariant detection, we need multiple executions of a statement to generalize the values (statistical support)

High level architecture



Phases

- Program instrumentation
  - Tells which variables to watch at what program points
- The inference step
  - Tests possible invariants against values captured for instrumented variables
  - Reported properties are the ones that
    - Are satisfied over all the values of a variable
    - Are statistically justified
    - Are not over unrelated variables
    - Are not implied by other reported invariants

Invariant detection

- The execution of an instrumented program stores the values of all variables at an interesting program point

- Suppose x, y, and z are in scope at a watched point
- We test all invariants (constructed from a template library) on x, y, z
  - All unary invariants checked for x, y, and z
  - All binary invariants checked for <x, y>, <x, z>, and <y, z>
  - All ternary invariants checked for <x, y, z>
  - It stops at ternary tuples

- Each invariant is checked on each trace
  - The check is over concrete values – no theorem proving, etc. – is cheap
- If any trace violates the invariant, it is not a correct invariant

Example – increment

```
int inc(int *x, int y)
  *x += y;
  return *x;
```

Watched point: end of the procedure

⟨	orig(x),	orig(*x),	orig(y),	x,	*x,	y,	return	⟩
⟨	4026527180,	2,	1,	4026527180,	3,	1,	3	⟩
⟨	4026527180,	3,	1,	4026527180,	4,	1,	4	⟩
⟨	146204,	13,	1,	146204,	14,	1,	14	⟩
⟨	4026527180,	4,	1,	4026527180,	5,	1,	5	⟩
⟨	146204,	14,	1,	146204,	15,	1,	15	⟩
⟨	4026527180,	5,	1,	4026527180,	6,	1,	6	⟩
⟨	4026527180,	6,	1,	4026527180,	7,	1,	7	⟩

What are some potential invariants?

Example – increment

```
int inc(int *x, int y)
  *x += y;
  return *x;
```

Watched point: end of procedure

⟨	orig(x),	orig(*x),	orig(y),	x,	*x,	y,	return	⟩
⟨	4026527180,	2,	1,	4026527180,	3,	1,	3	⟩
⟨	4026527180,	3,	1,	4026527180,	4,	1,	4	⟩
⟨	146204,	13,	1,	146204,	14,	1,	14	⟩
⟨	4026527180,	4,	1,	4026527180,	5,	1,	5	⟩
⟨	146204,	14,	1,	146204,	15,	1,	15	⟩
⟨	4026527180,	5,	1,	4026527180,	6,	1,	6	⟩
⟨	4026527180,	6,	1,	4026527180,	7,	1,	7	⟩

Invariants: x = orig(x), y = orig(y) = 1, $*$x = orig($*$x) + 1, and return = $*$x

Sample invariant templates

- For any variable
  - Constant value: x = a
  - Uninitialized: x = uninit (x is never set)
  - Small value set: x \in {a, b, c}
- For single numeric variable
  - Range limit: x >= a, x <= b
  - Nonzero: x != 0
  - Modulus: x mod b = a
- For two numeric variables
  - Linear relationship: y = ax + b
  - Ordering comparison: x < y, x <= y, x != y, ..
  - Functions: y = fn(x) (e.g. fn = absolute value, negation, bitwise complement)
  - All single-variable invariants over (x+y)

Sample invariant templates

- For 3 numeric variables
  - Linear relationship: z = ax + by + c
  - Functions: z = fn(x, y) (e.g. fn = min, max, multiplication, and, etc.)
- For a sequence variable (array)
  - Range: min and max of the sequence: a <= x[i] <= b
  - Element ordering: elements are non-decreasing, equal, non-increasing
- For two sequences
  - Linear relationship elementwise: y = ax + b
  - Subsequence: x is a subsequence of y
  - Reversal: x is the reverse of y
- For a sequence and a numeric variable
  - Membership: i \in s

How to instantiate the templates?

- Linear relationships like $x = ay + bz + c$ with a, b, c unknown
  - is instantiated by picking 3 tuples of values and computing a, b, c
- $x = a \pmod{b}$
  - is done by computing greatest common divisor of ($x1 – x2$) to get b (for different values x1 and x2 of x)
- $x < b$
  - is computed by updating b as more samples are seen

- Example?

Why these invariants?

- Based on users' programming and specification experience
- The list is built incrementally over time
- Not only added more invariants, but also removed the less useful ones

- Again, more invariants means longer runtime

- Users can add their own general and domain-specific invariants
  - Domain-specific: if a data structure is a tree

Output

- Functional invariants
  - Depends only on the code for a particular data structure or function
  - The invariant is universally true for any use of that entity
- Usage properties
  - Result from specific usage of a data structure or function
  - Depend on the context of use and the test suite

- Is this a true distinction?
  - Because Daikon operates on test suites, it cannot distinguish between these classes
  - Programmers cannot distinguish the two easily either because a sound pre-condition may be true only because the callers respect that

Experiments – rediscovery of formal specs

- Daikon can distinguish between
  - Preconditions (hold at beginning of a procedure)
  - Post-conditions (hold at the exit point of a procedure)
  - Rep invariants (hold both at the entry and the exit points of all procedures)
  - Loop invariants (hold at the beginning of each iteration of a loop)

- Daikon was applied to a set of textbook programs with formal pre/post conditions and loop invariants
  - All programs are small
    - Examples: searching, sorting, etc.
  - Formal spec was removed from the programs
  - A simple test suite was built
  - Daikon reported all those formal properties

Example – add array elements

$$
\begin{array}{l}
i, s := 0, 0; \\
\textbf{do } i \neq n \rightarrow \\
\qquad i, s := i + 1, s + b[i] \\
\textbf{od}
\end{array}
$$

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

- 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                          100 samples
   N = size(B)                            (7 values)
   N in [7..13]                           (7 values)
   B                                      (100 values)
     All elements in [-100..100]          (200 values)

15.1.1:::EXIT                           100 samples
   N = I = orig(N) = size(B)              (7 values)
   B = orig(B)                            (100 values)
   S = sum(B)                             (96 values)
   N in [7..13]                           (7 values)
   B                                      (100 values)
     All elements in [-100..100]          (200 values)

15.1.1:::LOOP                           1107 samples
   N = size(B)                            (7 values)
   S = sum(B[0..I-1])                     (452 values)
   N in [7..13]                           (7 values)
   I in [0..13]                           (14 values)
   I <= N                                 (77 values)
   B                                      (100 values)
     All elements in [-100..100]          (200 values)
   B[0..I-1]                              (985 values)
     All elements in [-100..100]          (200 values)
```

- Entry invariants = preconditions
- The invariant N = size(B) is important, but missing from the hand-written spec

- Exit invariant = post-condition
- S = sum(B) is important
- No side effects on B and N

•Loop invariant:
- S = sum(B[0..I-1])

- Invariants give info about the test suite: N in [7..13] that can be used to improve the suite

Boxes represent the invariants that give the original formal spec

Application – program modification

- A case where inferred invariants were of substantial assistance to programmers
- “Replace” program:
  - Takes as input, a string, a regular expression and a replacement string
  - Outputs the input string with all occurrences of the regular expression changed to the replacement string
  - Is 563 LOC with 21 procedures in C
  - No comments or documentation
  - Decided to extend the language of the regular expression
  - Ran the code on 100 tests randomly selected from a suite
  - Daikon produced invariants at the entry and exit of each procedure
  - Two programmers started changing the program
    - Used invariants to make sure they understood the code correctly
    - They found a bug in the original code, represented by an unexpected invariant
    - Used Daikon on the changed code and compared the new invariants with the old ones to ensure lack of unintended changes

Improving invariants

- Just applying the templates doesn’t produce the desired invariants, and produces some unnecessary ones
- An invariant is relevant if it helps a programmer in his task
- The notion is highly dependent on Daikon’s developers experience

- To improve the relevance of reported invariants:
  - To add desired invariants
    - Add implicit values
    - Exploit unused polymorphism
  - To eliminate undesired invariants
    - Perform statistical confidence checks
    - Suppress redundant invariants
    - Limit which variables are compared to each other

1. Implicit values

- Some properties may be over entities not explicitly stored in program variables
  - Size of a data structure
  - Largest value of a data structure
  - Cyclicity of a data structure

- Daikon introduces “derived” variables to represent such entities
  - They are introduced at inference stage because their value can be determined from any trace
- So then ordinary invariant detection can report relationships involving these entities
- But
  - May slow down Daikon because now we have many more potential invariants
  - Inevitably, it increases the number of irrelevant invariants reported

Implicit values

- Derived variables for a sequence s (array)
  - Length: size(s)
  - Extreme elements: s[0], s[1], s[size(s)-1], s[size(s)-2]
    - To accommodate for header nodes, etc.
- For numeric sequence s
  - Sum: sum(s)
  - Minimum element: min(s)
  - Maximum element: max(s)
- For a sequence s and a numeric variable I
  - S[i], s[i-1]
  - Subsequence: s[0..i], s[0..i-1]
- For procedure invocations:
  - Number of calls in this trace so far
- User can add new derived variables

A few points

- Introducing new derived variables can be done recursively
  - a → size(a), then b, size(a) → b[size(a)-1]
  - Default recursion depth is set to 2
- Derived vars are introduced if previous invariants show they're sensible
  - This requires interleaving invariant detection and variable derivation
  - Introducing derived vars first and then invariant detection doesn't work
  - Derived variables are not introduced until invariants are computed over existing variables
  - Example for a sequence s,
    - Size(s) is introduced first, invariants are computed, then more sequence-based vars may be generated
    - If j >= size(s), then we won't create derived variable a[j]
  - Tautologies are not reported
    - i = size(s[0..i-1])
- Any time two vars are shown equal, one is canonically chosen and the other one is dropped from the set of variables

Example revisited – add array elements

```
15.1.1:::ENTER                  100 samples
  N = size(B)                     (7 values)
 [N in [7..13]]                   (7 values)
  B                               (100 values)
    All elements in [-100..100]   (200 values)

15.1.1:::EXIT                   100 samples
  N = I = orig(N) = size(B)       (7 values)
  B = orig(B)                     (100 values)
 [S = sum(B)]                     (96 values)
  N in [7..13]                    (7 values)
  B                               (100 values)
    All elements in [-100..100]   (200 values)

15.1.1:::LOOP                   1107 samples
  N = size(B)                     (7 values)
 [S = sum(B[0..I-1])]             (452 values)
  N in [7..13]                    (7 values)
 [I in [0..13]]                   (14 values)
 [I <= N]                         (77 values)
  B                               (100 values)
    All elements in [-100..100]   (200 values)
  B[0..I-1]                       (985 values)
    All elements in [-100..100]   (200 values)
```

Boxes represent the invariants that give the original formal spec

2. Polymorphism elimination

- What is the difference between polymorphism and generics?
- Variables declared as any polymorphic type (base class) usually contain a single type at runtime
  - A polymorphic list can be used for a list of integers
  - We like to infer invariants like list is sorted, but is not defined for list(object)
- Daikon uses the declared type (base type)
  - Because instrumentation is done up-front statically
  - That’s when we decide what to monitor
  - But can’t examine fields specific to the runtime type

Polymorphism elimination

- Two-pass solution
  - First pass watches base-class fields, object id, its runtime class
  - If Daikon detects invariants over the run-time class (e.g. if o != null then o.class = a specific class), then the user can add a comment with a more specific refined type
  - A second pass of instrumentation and invariant detection works on the refined type. Accesses fields of that type.
    - Sound if program runs over the same inputs, and is deterministic
    - Ow, exceptions might be thrown during code runs, and Daikon catches them

Example

Declarations:

```
class LinkedList { ListNode header; ... }
class ListNode { Object element;
                 ListNode next; ... }
class MyInteger { int value; ... }
```

Sample code:

```
LinkedList myList = new LinkedList();
for (int i=1; i<=10; i++)
  myList.add(new MyInteger(i));
```

LinkedList object invariant reported by Daikon:

header.closure(next).element.value is sorted by $\leq$

```
class ListNode { /*refined_type: MyInteger*/ Object element;
                 ListNode next; ... }
```

For recursive fields (e.g. next), variable header.closure(next) is all objects reachable from header.

A field of a set of objects gives the set of values for that field in all objects