Dynamically Discovering Likely Program Invariants to Support Program Evolution

Michael Ernst, Jake Cockrell, William Griswold, David Notkin

Published in IEEE Transactions on Software Engineering, 2001

Juyoung Jeon
July 17 2018

Motivation

- Programs evolve, and they need to sustain some properties over changes
- By explicitly stating the specification inside program code, the one who modifies the code can understand the properties that must be preserved at the modification
- An invariant is a form of program specification that asserts that a certain predicate at a program location always evaluates to true when it gets executed
  - Invariants are embedded in the taret program code (e.g., assert statement)
  - Invariants can be used to protect the program from adversial changes that violate assumptions that the program correctly behaves

Ex. Invariants of CircularFifoQueue

```java
private transient E[] elements;
private transient int start = 0;
private transient int end = 0;
private transient boolean full = false;
private final int maxElements;

public boolean add(final E element) {
    if (null == element)
    throw new NullPointerException
    ( "Attempted to add null object to queue");
    if (isAtFullCapacity())
        remove();
    elements[end++] = element;
    if (end >= maxElements)
        end = 0;
    if (end == start)
        full = true ;
    return true ;
}
```

- elements.length <= maxElements
- full == true -> start == end

- start >=0
- start < maxElements
- end >=0
- end < maxElements

- element !=null
- isAtFullCapacity() == false
- full == false

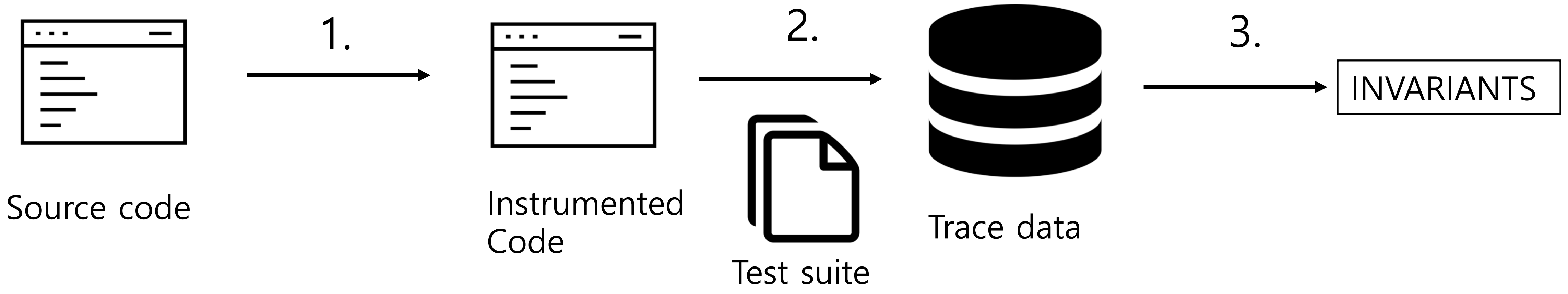
- full -> full
- element ∈ elements

Daikon

- Daikon is a tool that discovers invariants of a program by observing the target program executions
  - Input : a set of target Java classes and their test cases
  - Output
    1. Preprocessing (DynComp): decls file which is a list of grouped variables of the same abstract data type
    2. Invaraint extractor (Chicory): a list of invariants of given target classes
       - class invariants,
       - method pre-/post-condition
       - loop invariant

Workflow

1. Instrument source program to trace variables of interest
2. Run instrumented program over test suite
3. Check properties (infer invariants) over instrument variables and derived variables



Ex. Likely Invariants Inferrred by Daikon

```java
public boolean add(final E element) {
    if (null == element)
    throw new NullPointerException
    ( "Attempted to add null object to queue");
    if (isAtFullCapacity())
        remove();
    elements[end++] = element;
    if (end >= maxElements)
        end = 0;
    if (end == start)
        full = true ;
    return true ;
}
```

- True positive

  `this.start == org(this.start)`

  `return == true`

  `orig(arg0) == this.elements[orig(this.end)]`

- False positive

  `this.start <= orig(this.end)`

  `this.end one of { 0, 1, 2 }`

  `arg0 != null`

- c.f. False negative

  `this.full == false`

  `this.maxelements < 100`

Tutorial

- http://bit.ly/2LsC04Z

Invariant Inference Process Overview

1. Obtain a program trace which is a list of a variable name and its value at an execution point (method entry/exist points)
2. Create the initial set of likely invariant candidates by combinating programs variables according to the invariant templates.
3. Exam whether a candidate is consistent with the given program trace
   - throw away violating invariants from the candidate set
4. Exam whether a candidate is confident according to the given program trace
5. Return the consistent & confident invariants as the result

Step 1. Trace Data

- Trace the values of all variables in the scope at every program point along an execution

- Program points include every loop head, every procedure entry/exit point

Ex. CircularFifoQueue fields and add method

```java
private transient E[] elements;
private transient int start = 0;
private transient int end = 0;
private transient boolean full = false;
private final int maxElements;

public boolean add(final E element) {   P1
    if (null == element)
    throw new NullPointerException
    ( "Attempted to add null object to queue");
    if (isAtFullCapacity())
        remove();
    elements[end++] = element;
    if (end >= maxElements)
        end = 0;
    if (end == start)
        full = true ;
    return true ;   P2
}
```

Point	Assignment (variable-value map)
…	…
P1	this.elements = [1, 2, 3,], this.start = 0, this.end = 3, this.full = false, this.maxElement = 10 element = 4
P2	this.elements = [1, 2, 3,], this.start = 0, this.end = 3, this.full = false, this.maxElement = 10 element = 4, \result = true
…	…
P1	this.elements = [1, 2, 3], this.start = 0, this.end = 0, this.full = true, this.maxElement = 3, element = 4
P2	this.elements = [4, 2, 3,], this.start = 1, this.end = 1, this.full = true, this.maxElement = 3, element = 4, \result = true
…	…

Step 2. Inferring Invariants

- Detect invariant at a specific program execution point

- Create set of potential invariants over variables in the code and derived variables that are not explicitly stated in the code using given templates
  - The sets will be created for unary, binary, and tenary tuple of variables

Templates for inferring invariants (1/2)

- Invariants over variable
  - Any Variable
    - Constant value, uninitialized variable, small value set
  - Single numeric
    - Range limits, nonzero, modulus, non modulus
  - Two numeric
    - Linear relationship, ordering comparison, functions, invariants over x+y, invariants over x-y
  - Three numeric
    - Linear relationship, functions
  - Single sequence
    - Range (min, max, ordered), element ordering, invariants over all sequence elements
  - Two Sequence
    - Linear relationship, comparison, subsequence relationship, reverasal
  - A Sequence and a numeric
    - membership

Ex. CircularFifoQueue Inferring Invariants over variables

- Unary :
  - maxElements ==10
  - $0 \leq$ element ≤ 5
  - return ==true

- Binary:
  - orig(maxElements) == maxElements
  - orig(elements) != elements

- Tenary:
  - orig(end) == 3*end +2*maxElements +1

Templates for inferring invariants(1/2)

- Invariants over derived variables
  - Derived from any sequence
    - length (size)
    - Extremal elements (s[0], s[1], s[size(s)-1], …)
  - Derived from any numeric sequence
    - Sum
    - Minimum element
    - Maximum element
  - Derived from any sequence s and numeric variable i
    - Element at index
    - Subsequences (s[0 … i])
  - Derived from function invocations
    - Number of calls
- Invariants over derived variables are introduced in stages because computed invariants can derive other variables

Ex. CircularFifoQueue Inferring Invariants over derived variables

- Unary :
  - sum(elements) <20
  - min(elements) ==0
- Binary:
  - 0 $\leq$ size(elements) $<$ maxElements
  - size(orig(elements))< size(elements)

Step 3. Filter Valid Invariants

- Checks unary, binary, and tenary invariants for tuple of variables by examining each sample
  - Each set of possible invariants is tested against various combination of traced variables
- When sample not satisfying invariant is met, the invariant is thrown away and will not be checked for the rest of the samples

Ex. CircularFifoQueue fields and add method

```
private transient E[] elements;
private transient int start = 0;
private transient int end = 0;
private transient boolean full = false;
private final int maxElements;

public boolean add(final E element) {   P1
    if (null == element)
    throw new NullPointerException
    ( "Attempted to add null object to queue");
    if (isAtFullCapacity())
        remove();
    elements[end++] = element;
    if (end >= maxElements)
        end = 0;
    if (end == start)
        full = true ;
    return true ;   P2
}
```

Point	Assignment (variable-value map)
…	…
P1	this.elements = [1, 2, 3,], this.start = 0, this.end = 3, this.full = false, this.maxElement = 10 element = 4
P2	this.elements = [1, 2, 3,], this.start = 0, this.end = 3, this.full = false, this.maxElement = 10 element = 4, \result = true
…	…
P1	this.elements = [1, 2, 3], this.start = 0, this.end = 0, this.full = true, this.maxElement = 3, element = 4
P2	this.elements = [4, 2, 3,], this.start = 1, this.end = 1, this.full = true, this.maxElement = 3, element = 4, \result = true
…	…

Ex. CircularFifoQueue Inferring Invariants over variables

- Unary :
  - maxElements ==10
    - Falsified in the second sample when maxElements ==3
  - 0≤element ≤5
    - Passed for both samples
  - return ==true
    - Passed for both samples
- Binary:
  - start < end
    - Falsified for the second sample when start == end ==1
  - orig(elements) != elements
    - Passed for both samples
- Tenary:
  - orig(end) == 3*end +2*maxElements +1
    - Falsified in the first sample when end ==3, maxElements ==10

Ex. CircularFifoQueue Inferring Invariants over derived variables

- Unary :
  - sum(elements) <20
    - Passed by all samples
  - min(elements) ==0
    - Falsified in the first sample when min(element)==1
- Binary:
  - $0 \leq$ size(elements) $<$ maxElements
    - Passed by all samples
  - size(orig(elements))< size(elements)
    - Falsified in the second sample when size(orig(elements)) == size(elements) ==3

Step 4. Spurious Invariants

- Not all of unfalsified invariants are important / correct

- Correctness of generated invariants depend largely on the test suite

- Solution
  - Use of larger, complete test suites
  - Method for computing invariant confidence
  - Prune invariants logically implied by other invariants

Invariant Confidence

- Plausibility of an invariant
  - Inadequate number of samples of particular variable means patterns observed over the variable may be a coincidence
- Compute probability that such property appear by change in random input
- If probability is smaller than user specified confidence parameter, it is considered non coincidental and is reported

Ex. Computing Confidence for CircularFifoQueue

- Suppose that Daikon infers an invariant `maxElements!=0`
- Hypothesis: Daikon conjectures that the values of a non-negative integer `maxElements` are uniformly distributed over range between 0 and R
- P-value: the probability of missing the `maxElements` being 0 case in N samples: $\left(1 - \frac{1}{R}\right)^N$
- Report an inferred invariant only if the p-value does not exceed the given confidence level (e.g., 0.01)