DySy: Dynamic Symbolic Execution for Invariant Inference

C. Csallner – N. Tillmann – Y. Smaragdakis

Marc Egg
Invariant Inference

Object top() {
    if(Empty)
        return null;
    return theArray[topOfStack];
}

Invariant Inference Tool

postcondition: topOfStack = old topOfStack
class invariant: theArray != null
class invariant: topOfStack >= 0 && topOfStack < theArray.Length
Daikon

• First and most mature dynamic invariant inference tool

• Work flow
  – Instrumentation of all variables in scope of program
  – Execution of program
  – At each method entry / exit
    • Instantiation of invariant templates
    • Disqualification of inferred invariants which are refuted by an execution trace

• Invariant templates
  – Frequently used invariant patterns
Dynamic Invariant Inference – Problems

• Inferred invariants often
  – irrelevant
  – false
  – occasionally interesting but too simplistic
  – reflect the test suite

• Daikon's dubious invariants
  – theArray.getClass() != result.getClass()
  – topOfStack >> DEFAULT_CAPACITY == 0
DySy

• Solution proposed by authors
  – Invariant inference using dynamic symbolic execution

• Idea
  – Execute program symbolically in parallel to real execution
  – Record path condition
  – Use recorded path conditions to infer invariants

• DySy implements this idea
Symbolic Execution

- Replaces concrete inputs of a method with symbolic values

- Path condition
  - Accumulator of properties which the inputs must satisfy in order for an execution to follow the particular associated path
    - Explicit branches (control flow)
    - Implicit branches (exceptional behavior)
Symbolic Execution – Example

```c
int testme(int x, int y) {
    int prod = x * y;
    if(prod < 0) {
        throw new ArgumentException();
    }  
    if(x < y) { // swap them
        int tmp = x;
        x = y;
        y = tmp;
    }
    int sqry = y * y;
    return prod * prod - sqry * sqry;
}
```
Symbolic Execution – Example

- Path
  - 0-1-2-3-5-7-8

- Initial state
  - $x \rightarrow X$, $y \rightarrow Y$

- Final state
  - result $\rightarrow X*Y*X*Y - X*X*X*X$

- Path condition
  - $!(X * Y < 0) \&\& (X < Y)$
DySy – Algorithm

• Step 1: Path condition & final state discovery
  – New interpreter instance for every method call
  – Interpreter evolves symbolic state according to all subsequently executed instructions
    • Detection of purity of method
    • Pure methods used as logical variables in path conditions
    • Recursion treated as logical variables as well
      – result == ((i <= 1) \rightarrow 1) else i * fac(i-1)
  – Quadruple (method, pathCondition, result, finalState) recorded when method returns
DySy – Algorithm

● Step 2: Class invariant derivation
  – Computation of “class invariant candidates” of class $C$
    • Set of conjuncts $c$ of all recorded path conditions of all methods of $C$ where $c$
      only refers to the this argument
  – DySy checks which candidates are implied by all path conditions in
    the final states of all methods of $C$
    • Current implementation: DySy executes the test suite again and checks the
      candidates in the concrete final state of each call to a method of $C$
  – Class invariants used to simplify invariants of methods
DySy – Algorithm

- **Step 3: Pre- and postcondition computation**
  - Precondition of a method
    - Disjunction of its path conditions
  - Postcondition of a method
    - Conjunction of its path-specific postconditions
  - Path-specific post condition is an implication
    - Left hand side: path condition
    - Right hand side: Conjunction of equalities where each equality relates a logical variable to a term in the final state
DySy – Inference example

- **Path conditions**
  - !(x * y < 0) && (x < y)
  - !(x * y < 0) && !(x < y)

- **Precondition**
  - x * y >= 0

- **Postcondition**
  - result == (((x < y ) → x*y*x*y – x*x*x*x) else (x*y*x*y - y*y*y*y)

```java
int testme(int x, int y) {
    int prod = x * y;
    if(prod < 0) {
        throw new ArgumentException();
    }
    if(x < y) { // swap them
        int tmp = x;
        x = y;
        y = tmp;
    }
    int sqry = y * y;
    return prod * prod - sqry * sqry;
}
```
DySy – Loops

- Problem: enormous path conditions with straightforward symbolic execution

- **for** loops
  - Loop variables treated as symbolic values
  - Exit condition not recorded in path condition if loop body is entered
  - Symbolic conditions in loop body collapsed per-program-point with only the last value remembered

- Other kinds of loops are future work
DySy – Loop example

public int linSearch(int ele, int[] arr) {
    if(arr == null) {
        throw new ArgumentException();
    }
    for(int i = 0; i < arr.Length; i++) {
        if(ele == arr[i]) {
            return i;
        }
    }
    return -1;
}

• Postcondition (simplified)
  - !(ele == arr[i]) → result == -1 || ele == arr[i] → result == $i
DySy – Evaluation

• Comparison between DySy and Daikon using the StackAr benchmark
  – StackAr: Stack algebraic data type using an array
  – Benchmark used for case study in Daikon literature
  – Java implementation
  – Authors rewrote StackAr in C#

• Reference invariants hand-produced by human user
DySy – Results of evaluation

<table>
<thead>
<tr>
<th>Goal</th>
<th>Daikon</th>
<th></th>
<th>DySy</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Strict</td>
<td>Relaxed</td>
<td>Strict</td>
<td>Relaxed</td>
</tr>
<tr>
<td>Total</td>
<td>27</td>
<td>19</td>
<td>27</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 1 – Number of inferred reference invariants

- Strict count
  - Detection of deep object equality
  - Detection of full purity of method

- Relaxed count
  - Detection of reference equality
DySy – Results of evaluation

<table>
<thead>
<tr>
<th></th>
<th>Invariants</th>
<th>Unique subexpressions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Goal</td>
<td>Daikon</td>
</tr>
<tr>
<td>Total</td>
<td>27</td>
<td>138</td>
</tr>
</tbody>
</table>

Table 2 – Total number of inferred invariants and unique subexpressions

- Performance
  - Daikon: 9 seconds
  - DySy: 28 seconds
DySy – Quote

“We believe that this technique represents the future of dynamic invariant inference.”
DySy – Impact

- 35 citations (ACM Digital Library)
- Limited influence
- DySy not maintained anymore
DySy – Assessment

- As capable as Daikon but less verbose

- Many open issues
  - Ruling out invalid class invariant candidates inefficient
  - Large overhead due to symbolic execution
  - No support for loops except \textit{for} loops

- Quality of invariants heavily depends on the test suite

- Proven to work well only for this particular stack