Seminar in Software Engineering

DySy
Dynamic Symbolic Execution for Invariant Inference

April 28th 2009

Lukas Schwab
lschwab@student.ethz.ch
The authors

• Christoph Csallner
  - College of Computing, Georgia Tech

• Nikolai Tillmann
  - Microsoft Research

• Yannis Smaragdakis
  - CS Dept., University of Oregon
Motivation

• You get some code from someone

• It might be helpful to know

  (1) preconditions
  (2) postconditions
  (3) class invariants

```java
int foo(int x, int y) {
    int prod = x*y;
    if (prod <= 0)
        throw new Exception();
    if (x < y) {
        int tmp = x;
        x = y;
        y = tmp;
    }
    int sqry = y*y;
    return prod - sqry;
}
```
Dynamic invariant inference

Source code → Dynamic invariant inference system → Invariants

Test suite

Input

Output

Invariants = \{preconditions, postconditions, class invariants\}
Daikon

- **Daikon is the most widely used dynamic invariant inference system**
  - Available for C, C++, Eiffel, Java and Perl
- **Generates invariants by using predefined invariant templates, e.g.**
  - Constant: $x = 0$
  - Order: $x \leq y$
  - Linear: $x = a \times y + b$
  - User defined
Daikon: How does it work?

- Daikon proposes some invariants
- With each execution it disqualifies those invariants which evaluate to false

<table>
<thead>
<tr>
<th>Input x</th>
<th>Result r</th>
<th>Constant</th>
<th>Order</th>
<th>Linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f(1) = 2</td>
<td>r = 2</td>
<td>r &gt; x</td>
<td>r = 2*x + 0</td>
</tr>
<tr>
<td>2</td>
<td>f(2) = 4</td>
<td>r = 2</td>
<td>r &gt; x</td>
<td>r = 2*x + 0</td>
</tr>
<tr>
<td>-3</td>
<td>f(-3) = -6</td>
<td>r &gt; x</td>
<td></td>
<td>r = 2*x + 0</td>
</tr>
</tbody>
</table>

Inferred postcondition: \( r = 2^*x \)
Daikon: Problems

• Bad test suite $\rightarrow$ bad invariants
• Complicated invariants can not be inferred
  - User has to define additional templates
• Invariants are often irrelevant
• Invariants hold sometimes accidentally
Symbolic execution

• Execute program symbolically instead with concrete values

• Path condition: „Accumulation of properties which the input must satisfy in order for an execution to follow the particular associated path.“
Path condition: Example

```c
int foo(int x, int y) {
    int prod = x*y;
    if (prod <= 0)
        throw new Exception();
    if (x < y) {
        int tmp = x;
        x = y;
        y = tmp;
    }
    int sqry = y*y;
    return prod - sqry;
}
```

### Path condition

<table>
<thead>
<tr>
<th>Pos</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>(x \cdot y &gt; 0)</td>
</tr>
<tr>
<td>3</td>
<td>(x \cdot y &gt; 0 \land x \geq y)</td>
</tr>
</tbody>
</table>

### Value

<table>
<thead>
<tr>
<th>Variable</th>
<th>Concrete</th>
<th>Symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>prod</td>
<td>6</td>
<td>(x \cdot y)</td>
</tr>
<tr>
<td>sqry</td>
<td>4</td>
<td>(y \cdot y)</td>
</tr>
<tr>
<td>result</td>
<td>2</td>
<td>(x \cdot y - y \cdot y)</td>
</tr>
</tbody>
</table>
DySy: Overview

• DySy is a dynamic invariant inference system
  - Implemented in C#

• DySy performs a symbolic execution simultaneously with its concrete execution.
  - Take only the path of the concrete execution
  - Symbolic variables: (static) fields, parameters, variables and the result
```c
int foo(int x, int y) {
    int prod = x*y;
    if (prod <= 0)
        throw new Exception();
    if (x < y) {
        int tmp = x;
        x = y;
        y = tmp;
    }
    int sqry = y*y;
    return prod - sqry;
}
```

<table>
<thead>
<tr>
<th>Input</th>
<th>x=3, y=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path condition</td>
<td>x*y&gt;0 ∧ x&gt;=y</td>
</tr>
<tr>
<td>Result</td>
<td>x<em>y - y</em>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>x=2, y=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path condition</td>
<td>x*y&gt;0 ∧ x&lt;y</td>
</tr>
<tr>
<td>Result</td>
<td>x<em>y - x</em>x</td>
</tr>
</tbody>
</table>

Precondition: \((x*y>0 ∧ x\geq y) \lor (x*y>0 ∧ x<y)\)

Postcondition: if \((x*y>0 ∧ x\geq y)\) then \(x*y - y*y\)

else if \((x*y>0 ∧ x<y)\) then \(x*y - x*x\)
int foo(int x, int y) {
    int prod = x*y;
    if (prod <= 0)
        throw new Exception();
    if (x < y) {
        int tmp = x;
        x = y;
        y = tmp;
    }
    int sqry = y*y;
    return prod - sqry;
}

Precondition:  \( x*y > 0 \)
Postcondition: if (x>=y)  \( x*y - y^2 \)
               else          \( x*y - x^2 \)

<table>
<thead>
<tr>
<th>Input</th>
<th>x=3, y=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path condition</td>
<td>( x*y &gt; 0 \land x\geq y )</td>
</tr>
<tr>
<td>Result</td>
<td>( x*y - y^2 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>x=2, y=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path condition</td>
<td>( x*y &gt; 0 \land x&lt;y )</td>
</tr>
<tr>
<td>Result</td>
<td>( x*y - x^2 )</td>
</tr>
</tbody>
</table>
DySy Algorithm

• DySy uses another framework for symbolic execution

• The algorithm can be divided into three steps
Step 1: Run the code

- Execute a method
- Create state information
  - Quadruple (method, path condition, result, final state)
- Detect purity (no state change)
  - “array.Count == 0“ can be replaced with “array.IsEmpty“
- Abstract recursion
  - Prevents infinite invariants
Step 2: Class invariant derivation

• Define set of class invariant candidates
  - All path conditions which refer only to „this“

• Take all class invariant candidates, which are implied by all final path conditions.

• These are the class invariants
Step 3: Pre- & Postcondition

- Take all final path conditions
  - Preconditions: Conjunction of final path condition
    \[ \text{Precondition} = PC_1 \lor PC_2 \ldots \]
  - Postcondition: Implication with PC on the left and result under that PC on the right
    \[ \text{Postcondition} = (PC_1 \Rightarrow \text{Result}_1) \land (PC_2 \Rightarrow \text{Result}_2) \land \ldots \]

- Assume class invariants to be true and simplify
Loops: Problem

```
public int linSearch(int ele, int[] arr) {
    for (int i = 0; i < arr.Length; i++) {
        if (ele == arr[i])
            return i;
    }
    return -1;  // Not found
}
```

- Call `linSearch(3, {9,12,7})`
- Final path condition:
  ```
  arr.Length == 3 &&
  ```
- Precise but extremely useless
Loops: Solution

- Recognize loop variable $i$ as symbolic variable $\Rightarrow \ i$
- Collapse conditions inside the loop body
  - Just take the ones from the last run
- Ignore loop exit condition
  - $i < \text{arr.Length}$ vs $i \geq \text{arr.Length}$
Case Study

• Comparison of DySy with Daikon
  - Used Daikon's "StackAr" example

• Reference invariants were hand-produced by human user
  - Minimal
  - Easy to understand
## Case Study: Results (1)

<table>
<thead>
<tr>
<th>Goal</th>
<th>Recognized inv</th>
<th>Daikon</th>
<th>DySy</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Constructor</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>push</td>
<td>4</td>
<td>2 (4)</td>
<td>2 (4)</td>
</tr>
<tr>
<td>top</td>
<td>3</td>
<td>1 (3)</td>
<td>2 (3)</td>
</tr>
<tr>
<td>topAndPop</td>
<td>4</td>
<td>2 (4)</td>
<td>2 (4)</td>
</tr>
<tr>
<td>isEmpty</td>
<td>3</td>
<td>2 (3)</td>
<td>3</td>
</tr>
<tr>
<td>isFull</td>
<td>3</td>
<td>2 (3)</td>
<td>3</td>
</tr>
<tr>
<td>makeEmpty</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>27</td>
<td>19 (27)</td>
<td>20 (25)</td>
</tr>
</tbody>
</table>

- Higher is better
- In parentheses: Relaxed count (ignores deep equality of objects)
Case Study: Results (2)

<table>
<thead>
<tr>
<th></th>
<th>Unique sub-expr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Goal</td>
</tr>
<tr>
<td>Invariant Constructor</td>
<td>26</td>
</tr>
<tr>
<td>push</td>
<td>17</td>
</tr>
<tr>
<td>top</td>
<td>28</td>
</tr>
<tr>
<td>topAndPop</td>
<td>14</td>
</tr>
<tr>
<td>isEmpty</td>
<td>21</td>
</tr>
<tr>
<td>isFull</td>
<td>21</td>
</tr>
<tr>
<td>makeEmpty</td>
<td>9</td>
</tr>
<tr>
<td>isEmpty</td>
<td>13</td>
</tr>
<tr>
<td>makeEmpty</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td>89</td>
</tr>
</tbody>
</table>

- Lower is better
- Some sub-expressions are common across methods
- Daikon clearly infers to many invariants
• Both Daikon and DySy infer the required invariants well

• Daikon infers too many irrelevant invariants
  - this.theArray.getClass() != result.getClass()

• Bigger test suite would solve this problem
  - Huge test suite
  - Many invariant templates

• Symbolic execution is an improvement
My opinion (1)

- **DySy does not work as well as expected**
  - DySy fails to infer the class invariant „balance >= 0“ for the bank account example [See slide 25]

- **Usability extremely bad**
  - No integration in Visual Studio: You have to use the console, generated output hard to read
My opinion (2)

- Good idea to use symbolic execution
  - DySy gets more out of the code than Daikon
- Could be the future of invariant inference
public class BankAccount {
    private int _balance;

    public BankAccount() {
        _balance = 0;
    }

    public int Balance {
        get { return _balance; }
    }

    public void Deposit(int amount) {
        Balance += amount;
    }

    public void Withdraw(int amount) {
        if (Balance < amount) {
            throw new Exception("Not enough money");
        }
        Balance -= amount;
    }
}