Automatic Testing & Verification

Recap

Juan Pablo Galeotti, Alessandra Gorla,
Software Engineering Chair – Computer Science
Saarland University, Germany
Feb. 7th: Exam

- 30% projects (10% each)
  - At least 50% threshold for exam admittance
  - Groups of 2

- 70% final exam (see course schedule)
  - Closed-book
  - Allowed: one A4 page (both sides!)
Verification
  ▪ Against a specification
    ▪ It might be an implicit specification

Validation
  ▪ Does the system do what the user wants?
  ▪ Failures in specifications

Inference
  ▪ Discover some interesting properties about the program

Synthesis
  ▪ Create a new program: optimize (compiler), control (scheduler)

We will focus on verification and inference
Contract

A (formal) agreement between

Method M (callee)  Callers of M

Rights  Responsibilities  Rights  Responsibilities
Verifying Programs
Some JML Annotations

- @requires
- @ensures
- @signals
- @normal_behavior/exceptional_behavior
- @assert/assume
- @assignable/pure
- @loop_invariant/decreases
- @ghost
Program states

\[ \begin{align*}
  & \{ x \geq 4 \land y < -2 \} \\
  & \quad x := x + 1 \\
  & \{ x \geq 5 \land y < 0 \}
\end{align*} \]

WP

- \( X = 4 \) \( Y = -1 \)
- \( X = 4 \) \( Y = -1 \)
- \( X = 11 \) \( Y = -3 \)
- \( X \geq 4 \land y < -2 \)
- \( X = 5 \) \( Y = -1 \)
- \( X = 5 \) \( Y = -1 \)
- \( X = 5 \) \( Y = -1 \)
- \( X = 1 \) \( Y = -3 \)
- \( X = 5 \) \( Y = -1 \)
- \( X = 5 \) \( Y = -1 \)
- \( x := x + 1 \)
- \( x := x + 1 \)
- \( x := x + 1 \)
- \( x := x + 1 \)
- \( x := x + 1 \)
- \( X \geq 5 \land y < 0 \)
Calculating the Weakest Precondition

- $\text{WP}(\text{skip}, B) = \text{def } B$
- $\text{WP}(x := E, B) = \text{def } B[x \rightarrow E]$
- $\text{WP}(s_1 ; s_2, B) = \text{def } \text{WP}(s_1, \text{WP}(s_2, B))$
- $\text{WP}(\text{if } (E) \{ s_1 \} \text{ else } \{ s_2 \}, B) = \text{def }$
  
  $E => \text{WP}(s_1, B) \&\&$
  
  $!E => \text{WP}(s_2, B)$
Exercise!

- Complete the following Hoare Triple with the weakest precondition:

\[
\{???\}
\text{While}_<(x\geq 0, x) \ x > 0 \text{ do}
\begin{align*}
x &:= x - 1 \\
\text{EndWhile}
\end{align*}
\{x = 0\}
Problems with WP computation?

- Loop iterations!
- $\text{WP}_k(\text{while} \ (E) \ {S}, B)$
  - $\text{WP}_0(...) = \text{def} \ !E \Rightarrow B$
  - $\text{WP}_1(...) = \text{def} \ !E \Rightarrow B \ \& \ & E \Rightarrow \text{WP}(S,B)$
    
    $= \text{WP}_0(...) \ \& \ & E \Rightarrow \text{WP}(S,B)$
  - $\text{WP}_2(...) = \text{def} \ \text{WP}_1(...) \ \& \ & E \Rightarrow \text{WP}(S, \text{WP}_1(...))$
  - ....
  - $\text{WP}_{i+1}(...) = \text{def} \ \text{WP}_i \ \& \ & E \Rightarrow \text{WP}(S,\text{WP}_i(...))$
Dealing with loops

- Solutions:
  - **Unroll loops**: Verify a fixed set of execution traces
  - Add loop invariants to programs
We extend our WP definition for the new language constructs:

- $\text{WP (havoc } x, \ B) = \forall x. \ B$
- $\text{WP (assume } E, \ B) = E \Rightarrow B$
- $\text{WP (assert } E, \ B) = E \land B$
We transform loop code following this rule:

```
While_(I,T) E do S endwhile ==
assert I
havoc T
assume I
if (E) then
  S
assert
assume false
endif
```

Check Invariant hold at loop entry
Check loop body preserves Invariant
Object Invariant semantics

- An object invariant is a property that holds on every visible state of an object.
- What is a visible state?
  - The pre and post state of an invocation to a method of that object
- How to verify object invariants?
Modularity

- When we verify a method C.M() :
  - **Assume** that **ALL** invariants of all pre-existing objects hold at the method entry.
  - **Prove** that **ALL** invariants of all existing objects at the method exit hold
- When we invoke method C’.M’() from method C.M() :
  - **Prove** that **ALL** invariants of all pre-existing objects hold before executing the method.
  - **Assume** **ALL** invariants of all existing objects hold after executing the method.

**But this semantics is not modular**
Object invariants + ownership

- Object states:
  - Mutable
  - Valid
  - Comitted

- Each object might have a single owner
  - Ownership is a acyclic relation

- In order to change a field value the object must be in mutable state

- In order to make the object valid all owned objects have to be in valid state.

- The Committed state acts as a lock
Dataflow Analysis

- Over approximates all program behaviors
- Abstract State of behavior
- Dataflow direction: forward vs. backward
- May analysis vs. Must Analysis

<table>
<thead>
<tr>
<th>Direction \ U</th>
<th>\ U (MAY )</th>
<th>\ Cap (MUST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>reaching defs, zero analysis</td>
<td>available expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>live variable analysis</td>
<td>very busy expressions</td>
</tr>
</tbody>
</table>
(Forward) work-list algorithm

Compute $\text{out}[n]$ for each $n \in N$:

- Set $\text{out}[n] := \perp$
- Initialize $\text{work}.\text{add} = \{\text{entry}\}$

WHILE $\text{work}$ is not empty:

- Set $n := \text{work}.\text{pop}();$
- Set $\text{in}'[n] := \bigoplus \{ \text{out}[m] \mid m \in \text{pred}(n) \}$
- Set $\text{out}'[n] := \text{transfer}[n](\text{in}'[n])$
- IF !(\text{out}'[n] \subseteq \text{out}[n])
  
  FOR EACH $m \in \text{succ}(n)$ $\text{work}.\text{add}.add(m);$

- $\text{out}[n] := \text{out}'[n]$
- $\text{in}[n] := \text{in}'[n]$
Interprocedural Dataflow Analysis

- Analyze a program with many methods
- Strategies:
  - Build an interprocedural CFG
    - Inlining/Cloning
  - Assume/Guarantee
  - Context sensitivity
    - Inlining
    - Call string
    - Compute “summaries”
Code to generate inputs for:

```csharp
void CoverMe(int[] a)
{
    if (a == null) return;
    if (a.Length > 0)
        if (a[0] == 1234567890)
            throw new Exception("bug");
}
```

### Constraints to solve

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>Data</th>
<th>Observed constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>a!=null</td>
<td>null</td>
<td>a==null</td>
</tr>
<tr>
<td>a!=null &amp;&amp; ! (a.Length&gt;0)</td>
<td>{}</td>
<td>! (a.Length&gt;0)</td>
</tr>
<tr>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]!=1234567890</td>
<td>{123..}</td>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]!=1234567890</td>
</tr>
<tr>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]==1234567890</td>
<td>{123..}</td>
<td>a!=null &amp;&amp; a.Length&gt;0 &amp;&amp; a[0]==1234567890</td>
</tr>
</tbody>
</table>

Done: There is no path left.
Random Testing

- Create program inputs randomly
- Observe if the program behaves “correctly”
  - Using explicit contracts (pre & posts)
  - Implicitly: runtime undeclared exceptions
- Advantages:
  - Easy to implement
  - Good coverage if the test suite is big enough
Exhaustive Testing - Idea

- Generate all non-isomorphically valid inputs up to a given size.
- Use programmatic contracts to decide if an input is valid.
- Prune search space efficiently.
Genetic Algorithms

1. Initialize Population
2. Evaluate Population
3. While not done
   - Select parents
   - Recombine parents
4. Return best solution
Fitness

- **Approach level**
  - Number of control dependent edges between goal and chosen path
  - Approach = Number of dependent nodes - number of executed nodes
- **Branch distance**
  - Critical branch = branch where control flow diverged from reaching target
  - Distance to branch = distance to predicate being true / false
Some tools

- ESC/Java2, JMLForge
- Spec#
- Soot
- Javari/Plural
- Pex
- Korat
- EvoSuite