Test Case Generation for Object-Oriented Programs in CLP

Miguel Gómez-Zamalloa

joint work with Elvira Albert, Germán Puebla, José Miguel Rojas, Antonio Flores, etc

Complutense University of Madrid

First PROMETIDOS-CM Summer School

Madrid, September 19, 2011
Software testing accounts for a high percentage of the cost of software development.

Automation at different levels $\Rightarrow$ **Automatic generation of unit-tests**

**Unit-test = Test-data + test-oracle**

Different approaches to test-data generation (TDG):

- Random, black-box and glass-box
- Static vs. dynamic

We focus on glass-box, static TDG $\Rightarrow$ **Symbolic execution**

Symbolic execution:

- Execution with symbolic values $\Rightarrow$ constrained variables
- Indeterminism due to branching instructions involving unknown data
- Result: *Path conditions or equivalence classes of inputs*
Our Research on Software Testing

- Automatic TDG for object-oriented programs by symbolic execution
  - In particular Java (actually Java bytecode)
  - Keeping language independence as much as possible

- The PET tool (under development):
  - Automatic generation of JUnits
  - Integrated into Eclipse (heap and traces visualizer, etc)
Our Research on Software Testing

- **Automatic TDG for object-oriented programs by symbolic execution**
  - In particular Java (actually Java bytecode)
  - Keeping language independence as much as possible
- **The PET tool (under development):**
  - Automatic generation of JUnits
  - Integrated into Eclipse (heap and traces visualizer, etc)
- **Main aims:**
  - High degree of coverage with the less number of test-cases possible
  - Useful also for *program comprehension* (both source and bytecode)
  - Completeness w.r.t the coverage criterion
  - Flexibility w.r.t. the coverage criteria (e.g. *selective coverage criteria*)
The PET approach to TDG

- The PET approach: TDG by symbolic execution in CLP
  1. Compilation of the source language to CLP
  2. Symbolic execution in CLP
  3. Test cases backend (unit-tests and/or graphical representation)
The PET approach to TDG

- The PET approach: TDG by symbolic execution in CLP
  1. Compilation of the source language to CLP
  2. Symbolic execution in CLP
  3. Test cases backend (unit-tests and/or graphical representation)

- Strengths: Simple, generic and flexible

[Diagram showing the compilation process from PL to L to CLP Compiler, then to PCLP, then to TDG Engine, leading to Test Cases (CLP Format), which are then processed by Test case Viewer and L Code Generator, leading to Graphical Repr. and Unit Tests.]
Why Constraint-Logic Programming?

1. Translating O.O programs to CLP has been subject of active research
   - The first Futamura projection

\[ \text{LS Interpreter}_{(L_O)} \]

\[ P_{(L_S)} \rightarrow PE \rightarrow P_{(L_O)} \]
Why Constraint-Logic Programming?

Translating O.O programs to CLP has been subject of active research

- The *first Futamura projection*
- Interpretive (de)compilation of bytecode to CLP
Why Constraint-Logic Programming?

1. Translating O.O programs to CLP has been subject of active research
   - The first Futamura projection
   - Interpretive (de)compilation of bytecode to CLP

2. CLP handles symbolic execution natively
   - Backtracking mechanism
   - Constraints manipulation and solving
An Example

Java source code

```java
static int intExp(int a, int n) {
    if (n < 0) throw new ArithmeticException();
    else if ((a == 0) && (n == 0)) throw new ArithmeticException();
    else {
        int out = 1;
        for (; n > 0; n--) out = out*a;
        return out;
    }
}
```
intExp([[A,N],H_in],[Ret,H_out,EFlag]) :-
    N #>= 0,
    cond1(A,N,H_in,Ret,H_out,EFlag).

intExp([[A,N],H_in],[Ret,exc(R1)]) :-
    N #< 0, new(H_in,'ArithExc',R1,H'),
    init\AE([[ref(R1)],H'],[H_out,\_]).
intExp([[A,N],H_{in}],[Ret,H_{out},EFlag]) :-
N \#>= 0,
  cond1(A,N,H_{in},Ret,H_{out},EFlag).
intExp([[A,N],H_{in}],[\_,H_{out},exc(R1)]) :-
N \#< 0, new(H_{in},'ArithExc',R1,H'),
  init_{AE}([[\text{ref}(R1)],H'],[H_{out},\_]).

cond1(A,N,H1,Ret,H2,ok) :-
  A \#\approx 0, loopinit(A,N,H1,Ret,H2).
cond1(0,N,H1,Ret,H2,EFlag) :-
  cond2(0,N,H1,Ret,H2,ExFlag).
CLP-translated program

```
intExp([[A,N],H_in],[Ret,H_out,EFlag]) :-
    N #>= 0,
    cond1(A,N,H_in,Ret,H_out,EFlag).

intExp([[A,N],H_in],[-,H_out,exc(R1)]) :-
    N #< 0, new(H_in,'ArithExc',R1,H'),
    init^[AE]([[ref(R1)],H'],[H_out,-]).

cond1(A,N,H1,Ret,H2,ok) :-
    A #\= 0, loopinit(A,N,H1,Ret,H2).
cond1(0,N,H1,Ret,H2,EFlag) :-
    cond2(0,N,H1,Ret,H2,ExcFlag).
```

```
cond2(A,N,H1,Ret,H2,ok) :-
    N #\= 0, loopinit(A,N,H1,Ret,H2).
cond2(0,N,H1,Ret,H2,ExcFlag) :-
    throw(H1,H2,exc(R)).

throw(H1,H3,exc(R)) :-
    new(H1,'ArithExc',R,H2),
    init^[AE]([[ref(R)],H2],[H3,-]).
```
CFG and Bytecode

CLP-translated program

```prolog
intExp([[A,N],H_in],[Ret,H_out,EFlag]) :-
    N #>= 0,
    cond1(A,N,H_in,Ret,H_out,EFlag).
intExp([[A,N],H_in],[Ret,H_out,exc(R1)]) :-
    N #< 0,
    new(H_in,'ArithExc',R1,H_out),
    init_AE([[ref(R1)],H'],[H_out,...]).

cond1(A,N,H1,Ret,H2,ok) :-
    A \= 0,
    loopinit(A,N,H1,Ret,H2).
cond1(0,N,H1,Ret,H2,ok) :-
    loopinit(0,N,H1,Ret,H2).
cond1(A,N,H1,Ret,H2,ExcFlag) :-
    cond2(A,N,H1,Ret,H2,ExcFlag).

cond2(A,N,H1,H2,ok) :-
    N #\= 0,
    loopinit(A,N,H1,Ret,H2).
cond2(-0,H1,-H2,ExcFlag) :-
    throw(H1,H2,ExcFlag).
cond2(A,N,H1,Ret,H2,ok) :-
    N #\= 0,
    loopinit(A,N,H1,Ret,H2).
cond2(0,N,H1,Ret,H2,ok) :-
    loopinit(0,N,H1,Ret,H2).
cond2(-0,N,Out,Out) :-
    N #\=< 0.
cond2(A,N,Out,Ret) :-
    N #> 0,
```

M. Gómez-Zamalloa (UCM)
Finiteness of the Process and Coverage Criteria

\[ \text{TDG} = \text{Symbolic exec. } + \] + constraint solving
Finiteness of the Process and Coverage Criteria

\[ TDG = \text{Symbolic exec.} + \text{termination criterion} + \text{constraint solving} \]

- Finiteness of the process \( \Rightarrow \) Termination criterion.
Finiteness of the Process and Coverage Criteria

\[ \text{TDG} = \text{Symbolic exec.} + \text{termination criterion} + \text{constraint solving} \]

- Finiteness of the process $\Rightarrow$ Termination criterion.
- Coverage criteria $\Rightarrow$ depth-$k$ vs. block-$k$
Challenges and Contributions

1. Finiteness of the process and coverage criteria (✓ [LOPSTR’08])
2. Handle object-oriented features (✓ [ICLP’10])
   - Creation of heap-allocated data during symbolic execution
   - Inheritance, virtual invocations and aliasing
Challenges and Contributions

1. Finiteness of the process and coverage criteria (✓ [LOPSTR’08])
2. Handle object-oriented features (✓ [ICLP’10])
   ▶ Creation of *heap-allocated data* during symbolic execution
   ▶ *Inheritance, virtual invocations* and *aliasing*
3. Compositionality (✓ [LOPSTR’10])
   ▶ Separate symbolic execution at method level (SCC level) + composition
   ▶ Allows scaling and handling incomplete code and libraries
Challenges and Contributions

1. Finiteness of the process and coverage criteria (✓ [LOPSTR’08])
2. Handle object-oriented features (✓ [ICLP’10])
   ▶ Creation of heap-allocated data during symbolic execution
   ▶ Inheritance, virtual invocations and aliasing
3. Compositionality (✓ [LOPSTR’10])
   ▶ Separate symbolic execution at method level (SCC level) + composition
   ▶ Allows scaling and handling incomplete code and libraries
4. Usability ➞ The PET Tool (✓ [PEPM’10] and [WCRE’11])
   ▶ JML preconditions ➞ Crucial in practice
   ▶ Test-case visualizer (heaps and traces) ➞ Program comprehension
Challenges and Contributions

1. Finiteness of the process and coverage criteria (✓ [LOPSTR’08])
2. Handle object-oriented features (✓ [ICLP’10])
   - Creation of *heap-allocated data* during symbolic execution
   - *Inheritance*, *virtual invocations* and *aliasing*
3. Compositionality (✓ [LOPSTR’10])
   - Separate symbolic execution at method level (SCC level) + composition
   - Allows scaling and handling incomplete code and libraries
4. Usability ⇒ The PET Tool (✓ [PEPM’10] and [WCRE’11])
   - JML preconditions ⇒ Crucial in practice
   - Test-case visualizer (heaps and traces) ⇒ Program comprehension
5. Guided TDG ⇒ Towards finding specific paths
   - Resource-driven TDG (Preliminary work [LOPSTR’11])
Heap Allocation in Symbolic Execution

Let us consider the following simple example:

```
int m(A x) {
    if (x.f < 0) x.f = 0;
    return x.g;
}
```
Let us consider the following simple example:

```java
int m(A x) {
    if (x.f < 0) x.f = 0;
    return x.g;
}
```

Set of test-cases to achieve full branch coverage:

<table>
<thead>
<tr>
<th>In args</th>
<th>Input heap</th>
<th>Out</th>
<th>Output heap</th>
<th>EFlag</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ref(X)]</td>
<td>(X,ob('A', [field(f,-1), field(g,1)]))</td>
<td>1</td>
<td>(X,ob('A', [field(f,0), ...])</td>
<td>ok</td>
</tr>
</tbody>
</table>
Let us consider the following simple example:

```java
int m(A x) {
    if (x.f < 0) x.f = 0;
    return x.g;
}
```

Set of test-cases to achieve full branch coverage:

<table>
<thead>
<tr>
<th>In args</th>
<th>Input heap</th>
<th>Out</th>
<th>Output heap</th>
<th>EFlag</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ref(X)]</td>
<td>[X,ob('A', [field(f,-1),field(g,1)])]</td>
<td>1</td>
<td>[(X,ob('A', [field(f,0),...])</td>
<td>ok</td>
</tr>
<tr>
<td>[ref(X)]</td>
<td>[(X,ob('A', [field(f,1),field(g,1)])])</td>
<td>1</td>
<td>[(X,ob('A', [field(f,1),...])</td>
<td>ok</td>
</tr>
</tbody>
</table>
Let us consider the following simple example:

```
Trivial example. Java code
int m(A x) {
    if (x.f < 0) x.f = 0;
    return x.g;
}
```

Set of test-cases to achieve full branch coverage:

<table>
<thead>
<tr>
<th>In args</th>
<th>Input heap</th>
<th>Out</th>
<th>Output heap</th>
<th>EFlag</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ref(X)]</td>
<td>[(X,ob('A', [field(f,-1), field(g,1)]))]H</td>
<td>1</td>
<td>[(X,ob('A', [field(f,0),...])]</td>
<td>ok</td>
</tr>
<tr>
<td>[ref(X)]</td>
<td>[(X,ob('A', [field(f,1), field(g,1)]))]H</td>
<td>1</td>
<td>[(X,ob('A', [field(f,1),...])]</td>
<td>ok</td>
</tr>
<tr>
<td>[null]</td>
<td>H</td>
<td></td>
<td>[(0,ob('NullPointExc',[]))]H</td>
<td>exc(0)</td>
</tr>
</tbody>
</table>
CLP-Decompiled code

\[
m([\text{ref}(X)], G, H_{\text{in}}, H_{\text{out}}, EF) :- \text{get\_field}(H_{\text{in}}, X, A:f, F), \text{r1}([X, F], G, H_{\text{in}}, H_{\text{out}}, EF).
\]

\[
m([\text{null}], G, H_{\text{in}}, H_{\text{out}}, \text{exc}(\text{RExc})) :- \text{new\_object}(H_{\text{in}}, \text{'NullPointExc'}, \text{RExc}, H_{\text{out}}).
\]

\[
r1([X, F], G, H, H', EF) :- F < 0, \text{set\_field}(H, X, A:f, 0, H'), \text{r2}([X], G, H', H'', EF).
\]

\[
r1([X, F], G, H, H', EF) :- F >= 0, \text{r2}([X], G, H', EF).
\]

\[
r2([X], G, H, H, \text{ok}) :- \text{get\_field}(H, X, A:g, G).
\]

Symbolic execution tree

\[
m(A_{\text{in}}, A_{\text{out}}, H_{\text{in}}, H_{\text{out}}, EF)
\]
CLP-Decomplied code

\[
m([\text{ref}(X)], G, H_{in}, H_{out}, EF) :\text{ get_field}(H_{in}, X, A:f, F), \quad r1([X, F], G, H_{in}, H_{out}, EF).
\]

\[
m([\text{null}], G, H_{in}, H_{out}, \text{exc}(R_{Exc})) :\text{ new_object}(H_{in}, 'NullPointExc', R_{Exc}, H_{out}).
\]

\[
r1([X, F], G, H, H', EF) :\quad F \#< 0, \quad \text{set_field}(H, X, A:f, 0, H'), \quad r2([X], G, H', H'', EF).
\]

\[
r1([X, F], G, H, H', EF) :\quad F \#>= 0, \quad r2([X], G, H', H', EF).
\]

\[
r2([X], G, H, H, \text{ok}) :\quad \text{get_field}(H, X, A:g, G).
\]

Symbolic execution tree

\[
m(A_{in}, A_{out}, H_{in}, H_{out}, EF)
\]

\[
A_{in}=[\text{ref}(X)], \quad A_{out}=G
\]

\[
\text{get_field}(H_{in}, X, A:f, F)
\]

\[
H_{in}=[(X, ob('A', [field(f,F)]))].]
\]

\[
r1([X, F], G, H_{in}, H_{out}, EF)
\]
CLP-Decompiled code

\[ m([\text{ref}(X)], G, H_{in}, H_{out}, EF) :\text{ get\_field}(H_{in}, X, A:f, F), \text{ r1}([X, F], G, H_{in}, H_{out}, EF). \]

\[ m([\text{null}], G, H_{in}, H_{out}, \text{exc}(RExc)) :\text{ new\_object}(H_{in}, ‘\text{NullPointExc’}, RExc, H_{out}). \]

\[ \text{r1}([X, F], G, H, H’, EF) :\text{ F} \#< 0, \text{ set\_field}(H, X, A:f, 0, H’), \text{ r2}([X], G, H’, H’’, EF). \]

\[ \text{r1}([X, F], G, H, H’, EF) :\text{ F} \#>= 0, \text{ r2}([X], G, H, H’, EF). \]

\[ \text{r2}([X], G, H, H, \text{ok}) :\text{ get\_field}(H, X, A:g, G). \]

Symbolic execution tree

\[
\begin{align*}
\text{m}(A_{in}, A_{out}, H_{in}, H_{out}, EF) \\
A_{in} = [\text{ref}(X)], A_{out} = G \\
\text{get\_field}(H_{in}, X, A:f, F) \\
H_{in} = [(X, ob(‘A’, [field(f,F)]))] ] \\
\text{r1([X, F], G, H_{in}, H_{out}, EF)} \\
F < 0 \\
\text{set\_field}(H, X, A:f, 0, H’) \\
H’ = [(X, ob(‘A’, [field(f,0)]))] ] \\
\text{r2([X], G, H’, H_{out}, EF)}
\end{align*}
\]
CLP-Decompiled code

\[
m([\text{ref}(X)], G, H_{in}, H_{out}, \text{EF}) :\text{ get\_field}(H_{in}, X, A : f, F), \quad r1([X, F], G, H_{in}, H_{out}, \text{EF}).
\]

\[
m([\text{null}], G, H_{in}, H_{out}, \text{exc}(\text{RExc})) :\text{ new\_object}(H_{in}, '\text{NullPointExc}', \text{RExc}, H_{out}).
\]

\[
r1([X, F], G, H, H', \text{EF}) :\quad F < 0, \quad \text{set\_field}(H, X, A : f, 0, H'), \quad r2([X], G, H', H', \text{EF}).
\]

\[
r1([X, F], G, H, H', \text{EF}) :\quad F \geq 0, \quad r2([X], G, H, H', \text{EF}).
\]

\[
r2([X], G, H, H, \text{ok}) :\quad \text{get\_field}(H, X, A : g, G).
\]

Symbolic execution tree

\[
m(A_{in}, A_{out}, H_{in}, H_{out}, \text{EF})
\]

\[
A_{in}=\text{[ref}(X), \quad A_{out}=G
\]

\[
\text{get\_field}(H_{in}, X, A : f, F)
\]

\[
H_{in}=\{(X, \text{ob}(\text{'}A\text{'}',[\text{field}(f, F)]))-\}\downarrow
\]

\[
r1([X, F], G, H_{in}, H_{out}, \text{EF})
\]

\[
F < 0 \downarrow
\]

\[
\text{set\_field}(H, X, A : f, 0, H')
\]

\[
H'=\{(X, \text{ob}(\text{'}A\text{'}',[\text{field}(f, 0)]))-\}\downarrow
\]

\[
r2([X], G, H', H_{out}, \text{EF})
\]

\[
H_{out}=H', \quad \text{EF}=\text{ok} \downarrow
\]

\[
\text{get\_field}(H', X, A : g, G)
\]

\[
H_{out}=\{(X, \text{ob}(\text{'}A\text{'}',[\text{field}(f, 0)], \ldots)\}\downarrow
\]

\[
H_{in}=\{(X, \text{ob}(\text{'}A\text{'}',[\text{field}(f, F), \text{field}(g, G)]))-\}\downarrow
\]

\[
\text{true}
\]
CLP-Decompiled code

\[
m([\text{ref}(X)], G, H_{\text{in}}, H_{\text{out}}, EF) :- \text{get\_field}(H_{\text{in}}, X, A:f,F), r1([X, F], G, H_{\text{in}}, H_{\text{out}}, EF).
\]

\[
m([\text{null}], G, H_{\text{in}}, H_{\text{out}}, \text{exc}(RExc)) :- \text{new\_object}(H_{\text{in}}, \text{'NullPointExc'}, RExc, H_{\text{out}}).
\]

\[
r1([X, F], G, H, H', EF) :- F #< 0, \text{set\_field}(H, X, A:f, 0, H'), r2([X], G, H', H', EF).
\]

\[
r1([X, F], G, H, H', EF) :- F #>= 0, r2([X], G, H, H', EF).
\]

\[
r2([X], G, H, H, \text{ok}) :- \text{get\_field}(H, X, A:g, G).
\]

Symbolic execution tree

\[
m(A_{\text{in}}, A_{\text{out}}, H_{\text{in}}, H_{\text{out}}, EF)
\]

\[
A_{\text{in}}=[\text{ref}(X)], A_{\text{out}}=G
\]

\[
\text{get\_field}(H_{\text{in}}, X, A:f,F)
\]

\[
H_{\text{in}}=[[X,\text{ob}'A',[\text{field}(f,F)]-]]\]

\[
r1([X, F], G, H_{\text{in}}, H_{\text{out}}, EF)
\]

\[
F<0
\]

\[
\text{set\_field}(H, X, A:f, 0, H')
\]

\[
H'=[[X,\text{ob}'A',[\text{field}(f,0)]-]]
\]

\[
r2([X], G, H', H_{\text{out}}, EF)
\]

\[
H_{\text{out}}=H', \text{EF}=\text{ok}
\]

\[
\text{get\_field}(H_{\text{in}}, X, A:g, G)
\]

\[
H_{\text{in}}=[[X,\text{ob}'A',[\text{field}(f,F),\text{field}(g,G)]-]]
\]

\[
\text{true}
\]

\[
H_{\text{out}}=[[X,\text{ob}'A',[\text{field}(f,0),\ldots]]\]

\[
H_{\text{in}}=[[X,\text{ob}'A',[\text{field}(f,F),\text{field}(g,G)]-]]
\]

\[
\text{true}
\]
CLP-Decomplied code

\[
\begin{align*}
m([\text{ref}(X)], G, H_{in}, H_{out}, EF) & : \text{get}_\text{field}(H_{in}, X, A:f, F), \ r1([X, F], G, H_{in}, H_{out}, EF). \\
m([\text{null}], G, H_{in}, H_{out}, \text{exc}(RExc)) & : \text{new}_\text{object}(H_{in}, 'NullPointExc', RExc, H_{out}). \\
r1([X, F], G, H, H', EF) & : F < 0, \ \text{set}_\text{field}(H, X, A:f, 0, H'), \ r2([X], G, H', H', EF). \\
r1([X, F], G, H, H', EF) & : F \geq 0, \ r2([X], G, H', H', EF). \\
r2([X], G, H, H, \text{ok}) & : \text{get}_\text{field}(H, X, A:g, G).
\end{align*}
\]

Symbolic execution tree

\[
\begin{align*}
m(A_{in}, A_{out}, H_{in}, H_{out}, EF) \\
A_{in}=[\text{ref}(X)], & \ A_{out}=G \\
A_{in}=[\text{null}], & \ EF=\text{exc}(0) \\
\text{get}_\text{field}(H_{in}, X, A:f, F) & \quad \text{new}_\text{object}(H_{in}, 'NPE', 0, H_{out}) \\
H_{in}=[(X, \text{ob}(A', [\text{field}(f, F)]))] & \quad H_{out}=[(0, \text{ob}(NPE, [])|H_{in}]
\end{align*}
\]

\[
\begin{align*}
r1([X, F], G, H_{in}, H_{out}, EF) & : F < 0 \\
H'=[(X, \text{ob}(A', [\text{field}(f, 0)]))] & \quad H_{out}=H_{in}, \ EF=\text{ok} \\
r2([X], G, H_{in}, H_{out}, EF) & : \text{get}_\text{field}(H_{in}, X, A:g, G) \\
H_{out}=H', \ EF=\text{ok} & \quad H_{in}=[(X, \text{ob}(A', [\text{field}(f, F), \text{field}(g, G)]))] \\
\text{get}_\text{field}(H', X, A:g, G) & \quad \text{true} \\
H_{out}=[(X, \text{ob}(A', [\text{field}(f, 0)], \ldots)] & \quad H_{in}=[(X, \text{ob}(A', [\text{field}(f, F), \text{field}(g, G)]))] \\
\text{true}
\end{align*}
\]
Aliasing in Symbolic Execution of OO Programs

Let us consider the following simple example:

Example. Java code

```java
void m(A x, B y) {
    // B subclass of A
    if (x.f < 0) y.f = 0;
    if (x.f == 0) y.f = -1;
}
```
Aliasing in Symbolic Execution of OO Programs

- Let us consider the following simple example:

  
  **Example. Java code**

  ```java
  void m(A x,B y){ // B subclass of A
    if (x.f < 0) y.f = 0;
    if (x.f == 0) y.f = -1;
  }
  ```

- Set of test-cases to achieve full branch coverage:

<table>
<thead>
<tr>
<th>In args</th>
<th>Input heap</th>
<th>Output heap</th>
<th>EFlag</th>
</tr>
</thead>
<tbody>
<tr>
<td>[r(X),null]</td>
<td>[(X,ob('A',[field(f,0)]))]</td>
<td>[(0,ob('NullPExc',[]))]</td>
<td>ex(0)</td>
</tr>
<tr>
<td>[r(X),null]</td>
<td>[(X,ob('A',[field(f,-1)]))]</td>
<td>[(1,ob('NullPExc',[]))]</td>
<td>ex(1)</td>
</tr>
<tr>
<td>[null, -]</td>
<td>H</td>
<td>[(2,ob('NullPExc',[]))]</td>
<td>ex(2)</td>
</tr>
</tbody>
</table>
Aliasing in Symbolic Execution of OO Programs

- Let us consider the following simple example:

  ```java
  void m(A x, B y)
  {
   // B subclass of A
   if (x.f < 0) y.f = 0;
   if (x.f == 0) y.f = -1;
  }
  ```

- Set of test-cases to achieve full branch coverage:

<table>
<thead>
<tr>
<th>In args</th>
<th>Input heap</th>
<th>Output heap</th>
<th>EFlag</th>
</tr>
</thead>
<tbody>
<tr>
<td>[r(X),null]</td>
<td>[(X,ob('A',[field(f,0)]))]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[r(X),null]</td>
<td>[(X,ob('A',[field(f,-1)]))]</td>
<td>[(0,ob('NullPExc',[]))]</td>
<td>ex(0)</td>
</tr>
<tr>
<td>[null,_]</td>
<td>H</td>
<td>[(1,ob('NullPExc',[]))]</td>
<td>ex(1)</td>
</tr>
<tr>
<td>[r(X),r(Y)]</td>
<td>[(X,ob('A',[field(f,-1)]))],(Y,ob('B',[field(f,-1)]))]</td>
<td>[(2,ob('NullPExc',[]))]</td>
<td>ex(2)</td>
</tr>
<tr>
<td>[r(X),r(Y)]</td>
<td>[(X,ob('A',[field(f,0)]))],(Y,ob('B',[field(f,0)]))]</td>
<td>... (Y,ob('B',[field(f,0)]))</td>
<td>ok</td>
</tr>
<tr>
<td>[r(X),_]</td>
<td>[(X,ob('A',[field(f,1)]))]</td>
<td>... (Y,ob('B',[field(f,1)]))</td>
<td>ok</td>
</tr>
</tbody>
</table>

M. Gómez-Zamalloa (UCM)
Aliasing in Symbolic Execution of OO Programs

- Let us consider the following simple example:

  Example. Java code

  ```java
  void m(A x, B y) {
      // B subclass of A
      if (x.f < 0) y.f = 0;
      if (x.f == 0) y.f = -1;
  }
  ```

- Set of test-cases to achieve full branch coverage:

<table>
<thead>
<tr>
<th>In args</th>
<th>Input heap</th>
<th>Output heap</th>
<th>EFlag</th>
</tr>
</thead>
<tbody>
<tr>
<td>[r(X),null]</td>
<td>[(X,ob('A',[field(f,0)]))]</td>
<td>[(0,ob('NullPExc',[]))]</td>
<td>ex(0)</td>
</tr>
<tr>
<td>[r(X),null]</td>
<td>[(X,ob('A',[field(f,-1)]))]</td>
<td>[(1,ob('NullPExc',[]))]</td>
<td>ex(1)</td>
</tr>
<tr>
<td>[null,_]</td>
<td>H</td>
<td>[(2,ob('NullPExc',[]))]</td>
<td>ex(2)</td>
</tr>
<tr>
<td>[r(X),r(Y)]</td>
<td>[(X,ob('A',[field(f,-1)])),(Y,ob('B',[field(f,-1)]))]</td>
<td>... (Y,ob('B',[field(f,0)]))</td>
<td>ok</td>
</tr>
<tr>
<td>[r(X),r(Y)]</td>
<td>[(X,ob('A',[field(f,0)])),(Y,ob('B',[field(f,0)]))]</td>
<td>... (Y,ob('B',[field(f,-1)]))</td>
<td>ok</td>
</tr>
<tr>
<td>[r(X),_]</td>
<td>[(X,ob('A',[field(f,1)]))]</td>
<td>[(X,ob('A',[field(f,1)]))]</td>
<td>ok</td>
</tr>
<tr>
<td>[r(X),r(X)]</td>
<td>[(X,ob('B',[field(f,0)]))]</td>
<td>[(X,ob('B',[field(f,0)]))]</td>
<td>ok</td>
</tr>
<tr>
<td>[r(X),r(X)]</td>
<td>[(X,ob('B',[field(f,-5)]))]</td>
<td>[(X,ob('B',[field(f,-1)]))]</td>
<td>ok</td>
</tr>
</tbody>
</table>
Challenges and Contributions

1. Finiteness of the process and coverage criteria (✓ [LOPSTR’08])
2. Handle object-oriented features (✓ [ICLP’10])
   ▶ Creation of *heap-allocated data* during symbolic execution
   ▶ *Inheritance, virtual invocations* and *aliasing*
3. Compositionality (✓ [LOPSTR’10])
   ▶ Separate symbolic execution at method level (SCC level) + composition
   ▶ Allows scaling and handling incomplete code and libraries
4. Usability ⇒ The PET Tool (✓ [PEPM’10] and [WCRE’11])
   ▶ JML preconditions ⇒ Crucial in practice
   ▶ Test-case visualizer (heaps and traces) ⇒ Program comprehension
5. Guided TDG ⇒ Towards finding specific paths
   ▶ Resource-driven TDG (Preliminary work [LOPSTR’11])
The Challenge...

Compositional TDG

\[ m(\ldots) \]
\[ \rightarrow \]
\[ \text{true} \]
\[ \rightarrow \]
\[ \ldots \]
\[ \rightarrow \]
\[ \phi \]
\[ \rightarrow \]
\[ \text{true} \]
\[ \rightarrow \]
\[ q(\text{Args}_{\text{in}}, \text{Args}_{\text{out}}, H_{\text{in}}, H_{\text{out}}, \text{EF}) \]
\[ \rightarrow \]

SymEx Tree for \( q \)

\[ \rightarrow \]
\[ \text{true} \]
\[ \rightarrow \]
\[ \text{true} \]
Compositional TDG

The Challenge...
Compositional TDG

The Challenge...

\[ m(\ldots) \]

\[ \text{true} \]

\[ q(\text{Args}_{\text{in}}, \text{Args}_{\text{out}}, H_{\text{in}}, H_{\text{out}}, EF) \]

\[ \implies \]

\[ \text{true} \]

\[ \phi \]

\[ \text{true} \]

\[ S_q \]

\[ \text{SymEx Tree for } q \]

\[ \text{true} \]

\[ \text{true} \]

\[ \text{true} \]

\[ \text{true} \]

\[ \text{true} \]

\[ \text{true} \]
The Challenge...

Compositional TDG

\[ m(\ldots) \rightarrow true \rightarrow \cdots \rightarrow \phi \rightarrow true \rightarrow q(\text{Args}_{\text{in}}, \text{Args}_{\text{out}}, H_{\text{in}}, H_{\text{out}}, \text{EF}) \leftarrow \cdots \rightarrow true \rightarrow \cdots \rightarrow true \rightarrow \phi \rightarrow true \rightarrow q(\text{Args}_{\text{in}}, \text{Args}_{\text{out}}, H_{\text{in}}, H_{\text{out}}, \text{EF}) \leftarrow \cdots \rightarrow true \rightarrow \cdots \rightarrow true \]

SymEx Tree for q

Summary for q
Top-down vs. Bottom-up

- **Top-down vs. Bottom-up**
  - Context-sensitive (top-down) symbolic execution
    - Pros: Only required information is computed
    - Cons: Reusability of summaries is not always possible (complexity of the associated operation)
  - Context-insensitive (bottom-up) symbolic execution
    - Pros: Composition can always be performed
    - Cons: Summaries can contain more test cases than necessary

M. Gómez-Zamalloa (UCM)
Top-down vs. Bottom-up

Context-sensitive (top-down) symbolic execution

**Pros.** Only required information is computed
**Cons.** Reusability of summaries is not always possible (complexity of the associated operation)
Top-down vs. Bottom-up

**Context-sensitive (top-down) symbolic execution**

- **Pros.** Only required information is computed
- **Cons.** Reusability of summaries is not always possible (complexity of the associated operation)

**Context-insensitive (bottom-up) symbolic execution**

- **Pros.** Composition can always be performed
- **Cons.** Summaries can contain more test cases than necessary
Challenges and Contributions

1. Finiteness of the process and coverage criteria (✓ [LOPSTR’08])
2. Handle object-oriented features (✓ [ICLP’10])
   - Creation of *heap-allocated data* during symbolic execution
   - *Inheritance, virtual invocations* and *aliasing*
3. Compositionality (✓ [LOPSTR’10])
   - Separate symbolic execution at method level (SCC level) + composition
   - Allows scaling and handling incomplete code and libraries
4. Usability ⇒ The PET Tool (✓ [PEPM’10] and [WCRE’11])
   - JML preconditions ⇒ Crucial in practice
   - Test-case visualizer (heaps and traces) ⇒ Program comprehension
5. Guided TDG ⇒ Towards finding specific paths
   - Resource-driven TDG (Preliminary work [LOPSTR’11])
The PET Tool

- Implemented in
  - Prolog $\Rightarrow$ compilation to CLP and TDG engine
  - Java $\Rightarrow$ Backend for eclipse plugin

- Some available options:
  - Computational trace (of bytecodes) associated to each test-case
  - Percentage of code-coverage
  - Automatic generation of junit

- Available coverage criteria (currently): depth-$k$ and block-$k$

- Interfaces:
  1. Command-line interface
  2. Web interface
  3. Eclipse plugin
     - Test case visualizer
     - Graphical representation of traces (interactive mode)

- Demo...
Challenges and Contributions

1. Finiteness of the process and coverage criteria (√ [LOPSTR’08])
2. Handle object-oriented features (√ [ICLP’10])
   - Creation of heap-allocated data during symbolic execution
   - Inheritance, virtual invocations and aliasing
3. Compositionality (√ [LOPSTR’10])
   - Separate symbolic execution at method level (SCC level) + composition
   - Allows scaling and handling incomplete code and libraries
4. Usability ⇒ The PET Tool (√ [PEPM’10] and [WCRE’11])
   - JML preconditions ⇒ Crucial in practice
   - Test-case visualizer (heaps and traces) ⇒ Program comprehension
5. Guided TDG ⇒ Towards finding specific paths
   - Resource-driven TDG (Preliminary work [LOPSTR’11])
Motivation and Selective Coverage Criteria

TDG = Symbolic exec. +

![Diagram of TDG with symbolic execution, termination criterion, and constraint solving]
Motivation and Selective Coverage Criteria

TDG = Symbolic exec. + termination criterion + constraint solving

- Coverage criteria: depth-$k$,
Motivation and Selective Coverage Criteria

\[ \text{TDG} = \text{Symbolic exec.} + \text{termination criterion} + \text{constraint solving} \]

Coverage criteria: \textbf{depth}-k, \textbf{block}-k \Rightarrow \text{Exhaustive coverage criteria}
Motivation and Selective Coverage Criteria

TDG = Symbolic exec. + termination criterion + constraint solving

- Coverage criteria: depth-$k$, block-$k$ ⇒ Exhaustive coverage criteria
- Selective coverage criteria: specific program point, specific exception, worst memory consumption (within a block-$k$ limit), . . .
Naive Approach to Selective TDG

Selective TDG (naive) = TDG + filtering of test-cases
Naive Approach to Selective TDG

Selective TDG (naive) = TDG + filtering of test-cases

- Paths in the symbolic execution tree can be labeled
Naive Approach to Selective TDG

Selective TDG (naive) = TDG + filtering of test-cases

- Paths in the symbolic execution tree can be labeled
- In practice, the CLP program is instrumented with traces
- Filtering is done by looking at the traces associated to the test-cases
Our Approach for Guided TDG

- **Challenge:** Avoid the generation of non-interesting paths
- **Idea:** Use the trace argument as an input to guide symbolic exec.

Guided TDG = Traces generator + guided symb. execs. + constr. solving

- Traces can be complete
- The different symbolic executions are independent of each other
- ▶ Can be performed in parallel and simplifies constraint solving

This allows going deeper in the tree
Our Approach for Guided TDG

- **Challenge:** Avoid the generation of non-interesting paths
- **Idea:** Use the trace argument as an input to guide symbolic exec.

Guided TDG = Traces generator + guided symb. execs. + constr. solving

- Traces can be complete
- The different symbolic executions are independent of each other
  - Can be performed in parallel and simplifies constraint solving
  - This allows going deeper in the tree
Our Approach for Guided TDG

- Challenge: Avoid the generation of non-interesting paths
- Idea: Use the trace argument as an input to guide symbolic exec.

Guided TDG = Traces generator + guided symb. execs. + constr. solving
Our Approach for Guided TDG

- **Challenge:** Avoid the generation of non-interesting paths
- **Idea:** Use the trace argument as an input to guide symbolic exec.

Guided TDG = Traces generator + guided symb. execs. + constr. solving

- Traces can be complete
Our Approach for Guided TDG

- Challenge: Avoid the generation of non-interesting paths
- Idea: Use the trace argument as an input to guide symbolic exec.

Guided TDG = Traces generator + guided symb. execs. + constr. solving

- Traces can be complete or partial
Our Approach for Guided TDG

- **Challenge:** Avoid the generation of non-interesting paths
- **Idea:** Use the trace argument as an input to guide symbolic exec.

Guided TDG = Traces generator + guided symb. execs. + constr. solving

- Traces can be complete or partial
- The different symbolic executions are independent of each other
  - Can be performed in parallel and simplifies constraint solving
- This allows going deeper in the tree
Trace Generators

- Trace generators are based on heuristics.
  - They produce *unfeasible* traces
  - Otherwise it would be the standard symbolic execution!
Trace Generators

- Trace generators are based on heuristics.
  - They produce *unfeasible* traces
  - Otherwise it would be the standard symbolic execution!
Trace Generators

- Trace generators are based on heuristics.
  - They produce *unfeasible* traces
  - Otherwise it would be the standard symbolic execution!

- **Soundness**: Every generated trace satisfies the property of interest
- **Completeness**: It generates all traces that satisfy the property
- **Effectiveness** is related with the number of generated unfeasible traces
Resource-driven TDG

- Let us instantiate guided TDG with resources
- Symbolic exec. is guided towards paths with an interesting resource consumption
- Application: Detect bugs related to an excessive consumption of resources
- Examples of resource policies:
  - Obtain test-cases whose resource consumption is larger than a threshold
  - Obtain a test-case for the execution with the highest resource consumption (within a limit)
- The approach is parametric on the type of resource (number of instructions, memory, etc)
- Trace generators could rely on the results of static analysis (COSTA)
Current Limitations and Further Challenges

- Support for floating-point numbers, bit operations, etc
- **Scalability** ⇒ One of the main issues in symbolic execution
  - JML preconditions ⇒ Reduction of the number of paths
  - Constraint-solving library (*clpfd* is now a bottle-neck)
  - Combination of concrete and symbolic execution
- Further challenge: Extension to handle **concurrent** programs
  - Current work towards CLP-based symbolic execution of **concurrent** objects