GENETIC PROGRAMMING FOR SOFTWARE TRANSPLANTS

IMAN HEMATI MOGHADAM
IMPLEMENTED APPROACH: OVERVIEW
Slicing:

- **Forward Slicing:**
  - Used to extract the implementation of the desired feature.

- **Backward Slicing:**
  - Used to extract how a desired feature is called.

The slicing is implemented using Wala.
CONSOLE OUTPUT FOR SLICING

1. NORMAL_RET CALLER: Node: < Application, Lc2/apps/klax/comp/ChuteArtist, handle(Lc2/fw/Notification;)V > Context: Everywhere[1]5 = invokevirtual< Application, Lc2/fw/Notification, name()Ljava/lang/String; > 2 @1 exception:4

2. NORMAL handle:8 = invokevirtual< Application, Ljava/lang/String, equals(Ljava/lang/Object;)Z > 5,6 @8 exception:7 Node: < Application, Lc2/apps/klax/comp/ChuteArtist, handle(Lc2/fw/Notification;)V > Context: Everywhere


4. NORMAL handle:12 = invokevirtual< Application, Ljava/lang/String, equals(Ljava/lang/Object;)Z > 5,10 @56 exception:11 Node: < Application, Lc2/apps/klax/comp/ChuteArtist, handle(Lc2/fw/Notification;)V > Context: Everywhere

5. PARAM_CALLER: Node: < Application, Lc2/apps/klax/comp/ChuteArtist, handle(Lc2/fw/Notification;)V > Context: Everywhere[29]12 = invokevirtual< Application, Ljava/lang/String, equals(Ljava/lang/Object;)Z > 5,10 @56 exception:11 v5

Difficult to translate the generated slices (which is in the form of WALA’s IR) back to source code.
**Code Generation:**

- **First Solution:**
  - Use a mapping between the slice’s statements and the source code’s line numbers
  - Not all lines of the slice represent complete Java statements, which leads to **syntactically** incorrect code

- **Second Solution:**
  - Transform the source code into an abstract syntax tree rather than using the original source file.
XML EXTRACTOR:

- Opportunistic use of XML technologies
  - Addressing and querying with XPath
  - Validating with schema languages such as XSD
XML Representation

- **srcML:**
  - A translator from code (C/C++/Java/C#) to srcML, and vice versa
  - A combination of source code (text) and AST information (tags)

- **srcML features:**
  - Preservation of all source code text (robust to code irregularities)
  - Easy to use and extend (compare it with AST)
  - Scalable translation
    - Translation speed over 25 KLOC/sec
**XPath Expressions:**

The GP algorithm is implemented using ECJ.
TREE BASED GP

Query1 = //unit[1]/class[1]/block[1]/field[3]/type[1]/specifier[1]

Query2 = //unit[2]/class[1]/block[1]/field[1]/type[1]/specifier[1]
XML Validator:
VALIDATING WITH SCHEMA LANGUAGE

- **XML Schema Definition (XSD)**
  - Defining the restriction on XML data structure, and used for validating XML files.

Diagram:
```plaintext
parser -> XSD -> Validator

- .XML passes through parser and XSD, then goes to Validator.
- If the structure validates, it is considered valid.
- If there is a mismatch, an error is indicated.
```
The current version supports 224 different kind of compiler errors.
Use also SDG in a case that quick fix has no suggestion.
FAULT LOCALIZATION & TEST CASE PURIFICATION:
SPECTRUM-BASED FAULT LOCALIZATION

- Automatically recommend a list of suspicious program elements for inspection based on testing results.

![Diagram of Program Spectra and Profile of an execution trace]

- Coverage information of one element \( (s_i) \) in all executions
- Correct or incorrect?

- \[ M \text{ spectra} \]
- \[ N \text{ parts} \]
- \[ e_1, e_2, \ldots, e_M \text{ errors} \]
SPECTRUM-BASED FAULT LOCALIZATION

- Different SBFL techniques are implemented:
  - Tarantula, Ochiai, Jaccard, and ...
  - No strong study of the effectiveness of various SBFL techniques in automated program repair.

- Missing code problem
  - When the logic error caused by missing some code, then no code available to be “suspected”.
  - Might be no problem in software transplant, but can be a problem in automated program repair?
TEST CASE PURIFICATION FOR IMPROVING SBFL

- Generate additional failing test cases to execute all assertions in a given failing test case [1].

```
1 Public class targetTest{
2   @Test
3   void t1(){
4     target t = new target();
5     int a=1;
6     assertEquals(2, t.inc(a));
7     int b=1;
8     assertEquals(0, t.dec(b));
9     int c=3;
10    assertEquals(1, t.dec_twice(c));
11   }
12}
```

---

**Test Case Purification for Improving SBFL**

- Generate additional failing test cases to execute all assertions in a given failing test case [1].

![Test Case Purification Diagram]

---

**Test Case Purification for Improving SBFL**

- Generate additional failing test cases to execute all assertions in a given failing test case [1].

Slicing

<table>
<thead>
<tr>
<th>Test case p2</th>
<th>Target Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Public class targetTest{</td>
<td>1 Public class target{</td>
</tr>
<tr>
<td>2   @Test</td>
<td>2   int inc(int n){</td>
</tr>
<tr>
<td>3     void p2(){</td>
<td>3     return ++n;</td>
</tr>
<tr>
<td>4       target t = new target();</td>
<td>4     };</td>
</tr>
<tr>
<td>5       int a=1;</td>
<td>5     int dec(int n){</td>
</tr>
<tr>
<td>6     assertEquals(2, t.inc(a));</td>
<td>6     return ++n;</td>
</tr>
<tr>
<td>7       int b=1;</td>
<td>7     };</td>
</tr>
<tr>
<td>8     assertEquals(0, t.dec(b));</td>
<td>8     int dec_twice(int n){</td>
</tr>
<tr>
<td>9       int c=3;</td>
<td>9     n = dec(n);</td>
</tr>
<tr>
<td>10    assertEquals(1, t.dec_twice(c));</td>
<td>10    return dec(n);</td>
</tr>
<tr>
<td>11    };</td>
<td>11    };</td>
</tr>
<tr>
<td>12}</td>
<td>12}</td>
</tr>
</tbody>
</table>

● means the statement is executed by the test case

- Fault localization Improved on **18** to **43%** of faults while performed worse on **1.3** to **2.4%** of faults [1].

XML Unparser:
## Experiments

<table>
<thead>
<tr>
<th>Subject</th>
<th>Type</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGAP</td>
<td>Donor</td>
<td>Marshalling Populations to XML</td>
</tr>
<tr>
<td>ECJ</td>
<td>Host</td>
<td></td>
</tr>
<tr>
<td><strong>TestCasePurification</strong></td>
<td>Donor</td>
<td>Test case Purification for improving Fault Localization</td>
</tr>
<tr>
<td><strong>GZoltar</strong></td>
<td>Host</td>
<td></td>
</tr>
<tr>
<td><strong>Zest</strong></td>
<td>Donor</td>
<td>Layout algorithms, which are currently missing in JGraphT</td>
</tr>
<tr>
<td><strong>JGraphT</strong></td>
<td>Host</td>
<td></td>
</tr>
<tr>
<td><strong>JEdit</strong></td>
<td>Donor</td>
<td>Auto indent, and syntax highlighting</td>
</tr>
<tr>
<td><strong>Ekit</strong></td>
<td>Host</td>
<td></td>
</tr>
</tbody>
</table>
CONCLUSION

- Present a GP Approach: used for both software transplant and program bug repair

- Advantages:
  - Based on XML and XPath
  - Fix compiler errors
  - Use Fault location technique and test case purification
Thank You