Lightweight Fault Localization Using Multiple Coverage Types

Raul Santelices
James A. Jones* Yanbing Yu
Mary Jean Harrold

Work supported by NSF CCF-0429117, CCF-0541049, and CCF-0725202, and by TCS Ltd.
Presenter supported by SIGSOFT CAPS
Previous Research and Limitations

Techniques based on coverage

- **Nearest Neighbors** [Renieris, Reiss ASE 2003]
- **SBI** [Liblit, Naik, Zheng, Aiken, Jordan PLDI 2005]
- **SOBER** [Liu, Yan, Fei, Han, Midkiff ESEC/FSE 2005]
- **Other derivative work** [Abreu, Zoeteweij, van Gemund TAIC-PART 2007] [Masri AUB-Tech.Rep. 2007]
Previous Research and Limitations

Techniques based on coverage

- **Nearest Neighbors** [Renieris, Reiss ASE 2003]
- **SBI** [Liblit, Naik, Zheng, Aiken, Jordan PLDI 2005]
- **SOBER** [Liu, Yan, Fei, Han, Midkiff ESEC/FSE 2005]

But...

- Operate on **individual** coverage types (e.g., statements, branches)
- No thorough **comparison** of the effectiveness of these coverage types
In This Work

- Method and study comparing coverage types
  - Statements
  - Branches
  - Data dependencies (du-pairs)

- Method and study combining coverage types

- Method and study reducing overhead
  - Overhead of statements and branches is 10-20%
  - Overhead of du-pairs is 60-120%, but du-pair coverage can be inferred (approximately) from branch coverage
Outline

- Coverage-based fault localization
- Comparing coverage types
- Combining coverage types for effectiveness
- Inferring du-pair coverage (approximately) to reduce overhead
- Conclusion and future work
Coverage-based Fault Localization

Test suite
Runtime information
- entities executed (covered)
- passing/failing tests

Analysis
- computes suspiciousness of each entity

Intuition: entities executed mostly by failing test cases are more suspicious than entities executed mostly by passing tests

```c
mid() {
    int x, y, z, m;
    1: read("Enter 3 integers:");
    2: m = z;
    3: if (y<z)  
        4:   if (x<y)  
            5:     m = y;
        6:   else if (x<z)
            7:     m = y;
    8: else
    9:   if (x>y)  
     10:     m = z;
    11: else if (x>z)
     12:     m = x;
    13: print("Middle number is: ");
}
```
Coverage-based Fault Localization

```c
mid() {
    int x, y, z, m;
    read("Enter 3 integers:", x, y, z); // t1: 3,3,5
    m = z; // t2: 1,2,3
    if (y < z) {
        if (x < y) m = y; // t3: 3,2,2
        else if (x < z) m = y; // t4: 5,5,5
    } else {
        if (x > y) m = z; // t5: 1,1,4
        else if (x > z) m = x; // t6: 5,3,4
    }
    print("Middle number is:", m); // t7: 3,2,1
} // t8: 2,1,3

5,5,5 // t9: 5,4,2
5,2,6 // t10: 2,2,1
```

Pass/fail Status:
```
P P P P P F F F F F
```
Technique: Tarantula

\[ \text{suspiciousness}(s) = \frac{\text{failed}(s)}{\text{total failed}} + \frac{\text{passed}(s)}{\text{total passed}} \]

```
int x, y, z, m;
1: read("Enter 3 integers:", x, y, z);
2: m = z;
3: if (y < z)
4:   if (x < y)
5:     m = y;
6:   else if (x < z)
7:     m = y;
8: else
9:   if (x > y)
10:    m = z;
11:   else if (x > z)
12:    m = x;
13: print("Middle number is:", m);
}
```

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>t9</th>
<th>t10</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,3,5</td>
<td>1,2,3</td>
<td>3,2,2</td>
<td>5,5,5</td>
<td>5,1,4</td>
<td>5,3,4</td>
<td>3,2,1</td>
<td>2,1,3</td>
<td>5,4,2</td>
<td>5,2,6</td>
</tr>
</tbody>
</table>

Pass/fail Status | P | P | P | P | P | F | F | F | F | F |

\[ \text{susp}(1) = \frac{\frac{4}{6}}{\frac{4}{6} + \frac{4}{4}} = 0.50 \]
Technique: Tarantula

```java
int x, y, z, m;
1: read("Enter 3 integers: ", x, y, z);
2: m = z;
3: if (y<z)
4:   if (x<y) m = y;
5: else if (x<z) m = y;
6: else
7:   if (x>y) m = z;
8: else m = y;
9: else
10: if (x>y) m = z;
11: else if (x>z) m = x;
12: print("Middle number is: ", m);
}
```

Pass/fail Status

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>t9</th>
<th>t10</th>
<th>suspiciousness</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,3,5</td>
<td>1,2,3</td>
<td>3,2,2</td>
<td>5,5,5</td>
<td>1,1,4</td>
<td>5,3,4</td>
<td>3,2,1</td>
<td>2,1,3</td>
<td>5,4,2</td>
<td>5,2,6</td>
<td>0.50</td>
</tr>
</tbody>
</table>

\[
susp(7) = \frac{2}{4} + \frac{2}{6} = 0.60
\]
## Technique: Tarantula

The Tarantula method uses suspiciousness scores to identify suspicious lines of code. The suspiciousness is calculated as a ratio of the number of failing test cases to the total number of test cases. A high suspiciousness score indicates that a line of code is more likely to be responsible for a failing test case.

### Example Code
```c
int x, y, z, m;
1: read("Enter 3 integers:", x, y, z);
2: m = z;
3: if (y<z)  
   4:   if (x<y)  
   5:     m = y;  
   6:   else if (x<z)  
   7:     m = y;  
3: else  
   4:   if (x>y)  
   5:     m = z;  
   6:   else if (x>z)  
   7:     m = x;  
13: print("Middle number is:", m);
```

### Test Cases
<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>t9</th>
<th>t10</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,3,5</td>
<td>1,2,3</td>
<td>3,2,2</td>
<td>5,5,5</td>
<td>1,1,4</td>
<td>5,3,4</td>
<td>3,2,1</td>
<td>2,1,3</td>
<td>5,4,2</td>
<td>5,2,6</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

### Suspiciousness Scores
<table>
<thead>
<tr>
<th>Line</th>
<th>suspiciousness</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.50</td>
</tr>
<tr>
<td>4</td>
<td>0.43</td>
</tr>
<tr>
<td>5</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>0.50</td>
</tr>
<tr>
<td>7</td>
<td>0.60</td>
</tr>
<tr>
<td>8</td>
<td>0.60</td>
</tr>
<tr>
<td>9</td>
<td>0.60</td>
</tr>
<tr>
<td>10</td>
<td>0.75</td>
</tr>
<tr>
<td>11</td>
<td>0.00</td>
</tr>
<tr>
<td>12</td>
<td>0.00</td>
</tr>
<tr>
<td>13</td>
<td>0.50</td>
</tr>
</tbody>
</table>

### Pass/Fail Status
<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>t9</th>
<th>t10</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

### Explanation
- **Line 3**: The line with the highest suspiciousness score is line 3, with a score of 0.50. This indicates that this line of code is likely to be responsible for a failing test case.
- **Line 8**: Line 8 also has a high suspiciousness score of 0.60, indicating it is a likely candidate for further investigation.
- **Line 9**: Line 9 has a score of 0.60, suggesting it could be related to a failing test case.
- **Line 11**: Line 11 has the lowest score of 0.00, indicating it is unlikely to be responsible for a failing test case.
# Technique: Tarantula

## suspiciousness(s) = \[
\frac{\text{failed(s)}}{\text{total failed}} + \frac{\text{passed(s)}}{\text{total passed}}
\]

```c
int x, y, z, m;
1: read("Enter 3 integers:", x, y, z);
2: m = z;
3: if (y<z)
   4:   if (x<y) m = y;
   5: else if (x<z) m = y;
   6: else if (x>y) m = z;
   7: else m = x;
8: else
9:   if (x>y) m = z;
10:  m = z;
11:  else if (x>z) m = x;
12:  m = x;
13: print("Middle number is:", m);
}
```

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>t9</th>
<th>t10</th>
<th>suspicion</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,3,5</td>
<td>1,2,3</td>
<td>3,2,2</td>
<td>5,5,5</td>
<td>1,1,4</td>
<td>5,3,4</td>
<td>3,2,1</td>
<td>2,1,3</td>
<td>5,4,2</td>
<td>5,2,6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pass/fail Status: P P P F F F F F F
**Technique: Tarantula**

The suspiciousness of a statement is calculated as:

$$\text{suspiciousness}(s) = \frac{\text{failed}(s)}{\text{total failed}} + \frac{\text{passed}(s)}{\text{total passed}}$$

<table>
<thead>
<tr>
<th>Pass/fail Status</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>P P P P P P F F F F</td>
<td></td>
</tr>
</tbody>
</table>

```
int x, y, z, m;
1: read("Enter 3 integers:", x, y, z); // buggy; correct m=y
2: m = z;
3: if (y<z)  
   4:   if (x<y)  
      5:     m = y;
   6: else if (x<z)  
      7:     m = y;
3: else  
   4:   if (x>y)  
      5:     m = z;
   6: else if (x>z)  
      7:     m = x;
13: print("Middle number is:", m);
}
```

Cost metric: 1/13 statements (7.7%)
Comparing Multiple Coverage Types

**Statements**

```c
mid(): int x,y,z,m;
1. read(x,y,z);
2. m = z;
3. if (y<z)
4.   if (x<y)
5.     m = y;
6.   else if (x<z)
7.     m = x;
8.   else
9.   if (x>y)
10.  m = y;
11.  else if (x>z)
12.  m = x;
13. print(m);
```

**Branches**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EN</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

**DU-pairs**

<table>
<thead>
<tr>
<th>DU-pair</th>
<th>DU-pair Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Execution**

- **statement coverage**
- **branch coverage**
- **du-pair coverage**

**Tarantula**

- **statement scores, ranking**
- **branch scores, ranking**
- **du-pair scores, ranking**
Comparing Multiple Coverage Types

**Statements**

```plaintext
mid(): int x, y, z, m;
1. read(x, y, z);
2. m = z;
3. if (y<z)
4.   if (x<y)
5.     m = y;
6.   else if (x<z)
7.     m = x;
8. else
9.   if (x>y)
10.  m = y;
11. else if (x>z)
12.  m = x;
13. print(m);
```

**Execution**

- **Statement coverage**
- **Tarantula**
  - Statement scores, ranking

**Branches**

```
```

- **Branch coverage**
- **Tarantula**
  - Branch scores, ranking

**DU-pairs**

```
```

- **Du-pair coverage**
- **Tarantula**
  - Du-pair scores, ranking
Comparing Multiple Coverage Types

Statements

```c
mid(): int x, y, z, m;
1. read(x, y, z);
2. m = z;
3. if (y<z)
4.   if (x<y)
5.     m = y;
6. else if (x<z)
7.     m = x;
8. else
9.   if (x>y)
10.    m = y;
11.   else if (x>z)
12.    m = x;
13. print(m);
```

Branches

DU-pairs

Solution

Statement ranking for each type!
Branch, Du-pair Mappings

1. Gather branch (du-pair) coverage
2. Compute suspiciousness for each branch (du-pair)
3. Map branches (du-pairs) to statements
4. Transfer suspiciousness scores of branches (du-pairs) to corresponding statements
5. Rank statements using the transferred suspiciousness scores

Result: statement rankings based on branches (du-pairs)
Experiment 1

**Goal:** compare individual coverage types

**Setup**

- **Coverage tool:** DUA-Forensics [Santelices, Harrold ASE 2007]
  - Target language: Java bytecode
  - Monitors statements, branches, and du-pairs
- **Fault-localization tool:** Tarantula with Mapper
  - Measure of cost: % of statements visited to reach first faulty statement
### Experiment 1: Subjects

- Siemens suite translated to Java
- NanoXML
- XML-Security
- JABA
- Total: 14 subjects

<table>
<thead>
<tr>
<th>subject</th>
<th>LOC</th>
<th>tests</th>
<th>faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcas</td>
<td>131</td>
<td>1608</td>
<td>10</td>
</tr>
<tr>
<td>Tot_info</td>
<td>283</td>
<td>1052</td>
<td>10</td>
</tr>
<tr>
<td>Schedule1</td>
<td>290</td>
<td>2650</td>
<td>9</td>
</tr>
<tr>
<td>Schedule2</td>
<td>317</td>
<td>2710</td>
<td>7</td>
</tr>
<tr>
<td>Print_tokens1</td>
<td>478</td>
<td>4130</td>
<td>5</td>
</tr>
<tr>
<td>Print_tokens2</td>
<td>410</td>
<td>4115</td>
<td>10</td>
</tr>
<tr>
<td>NanoXML v1</td>
<td>3497</td>
<td>214</td>
<td>7</td>
</tr>
<tr>
<td>NanoXML v2</td>
<td>4009</td>
<td>214</td>
<td>7</td>
</tr>
<tr>
<td>NanoXML v3</td>
<td>4608</td>
<td>216</td>
<td>8</td>
</tr>
<tr>
<td>NanoXML v5</td>
<td>4782</td>
<td>216</td>
<td>7</td>
</tr>
<tr>
<td>XML-security v1</td>
<td>21613</td>
<td>92</td>
<td>7</td>
</tr>
<tr>
<td>XML-security v2</td>
<td>22318</td>
<td>94</td>
<td>7</td>
</tr>
<tr>
<td>XML-security v3</td>
<td>19895</td>
<td>84</td>
<td>2</td>
</tr>
<tr>
<td>JABA</td>
<td>37966</td>
<td>677</td>
<td>11</td>
</tr>
</tbody>
</table>

**TOTAL FAULTS** 107
Experiment 1: Results

Faults with most different costs (>1% between best and worst coverage type)

For all 107 faults:

<table>
<thead>
<tr>
<th>measure</th>
<th>statement</th>
<th>branch</th>
<th>du-pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>average cost</td>
<td>11.49%</td>
<td>10.24%</td>
<td>9.02%</td>
</tr>
<tr>
<td>standard dev.</td>
<td>16.25%</td>
<td>15.40%</td>
<td>12.04%</td>
</tr>
</tbody>
</table>
Experiment 1: Results

Faults with most different costs (>1% between best and worst coverage type)

For all 107 faults:

<table>
<thead>
<tr>
<th>measure</th>
<th>statement</th>
<th>branch</th>
<th>du-pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>average cost</td>
<td>11.49%</td>
<td>10.24%</td>
<td>9.02%</td>
</tr>
<tr>
<td>standard dev.</td>
<td>16.25%</td>
<td>15.40%</td>
<td>12.04%</td>
</tr>
</tbody>
</table>
Experiment 1: Results

For all 107 faults:

<table>
<thead>
<tr>
<th>measure</th>
<th>statement</th>
<th>branch</th>
<th>du-pair</th>
<th>ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>average cost</td>
<td>11.49%</td>
<td>10.24%</td>
<td>9.02%</td>
<td>6.35%</td>
</tr>
<tr>
<td>standard dev.</td>
<td>16.25%</td>
<td>15.40%</td>
<td>12.04%</td>
<td>9.27%</td>
</tr>
</tbody>
</table>
Experiment 1: Results

- Different faults are best localized by different types
- DU-pairs were best overall, but there is plenty of room for improvement

For all 107 faults:

<table>
<thead>
<tr>
<th>measure</th>
<th>statement</th>
<th>branch</th>
<th>du-pair</th>
<th>ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>average cost</td>
<td>11.49%</td>
<td>10.24%</td>
<td>9.02%</td>
<td>6.35%</td>
</tr>
<tr>
<td>standard dev.</td>
<td>16.25%</td>
<td>15.40%</td>
<td>12.04%</td>
<td>9.27%</td>
</tr>
</tbody>
</table>
Combining Coverage Types

**Statements**

```c
mid(): int x, y, z, m;
1. read(x, y, z);
2. m = z;
3. if (y<z)
4.   if (x<y)
5.     m = y;
6.   else if (x<z)
7.     m = y;  // m = x
8. else
9.   if (x>y)
10.  m = y;
11.  else if (x>z)
12.   m = x;
13. print(m);
```

**Branches**

```c
1
2
3 F T
4 F T
5 F T
6 T F
7
8 F T
9 F T
10 F T
11 T F
12 T F
13
```

**DU-pairs**

- EN:
- Fault:
- T:
- F:

```
// m = x
```
Combining Coverage Types

Statements

```c
mid(): int x, y, z, m;
1. read(x, y, z);
2. m = z;
3. if (y<z)
   4. if (x<y)
      5. m = y;
      6. else if (x<z)
         7. m = y; // m = x
   8. else
      9. if (x>y)
        10. m = y;
        11. else if (x>z)
            12. m = x;
        13. print(m);
```

Branches

```
EN 1 → 2 → 3
  3 F 4 → 5 → 6
    6 T F 7
      7 F 8
        8 T 9
          9 T F 10
            10 F 11
              11 T 12
                12 T F 13
```

DU-pairs

<table>
<thead>
<tr>
<th>stmt.</th>
<th>score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>.71</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>.71</td>
<td>2</td>
</tr>
</tbody>
</table>

Fault

5. `m = y;`
Combining Coverage Types

Statements
mid(): int x,y,z,m;
1. read(x,y,z);
2. m = z;
3. if (y<z)
4.   if (x<y)
5.     m = y;
6.   else if (x<z)
7.     m = y; // m = x
8. else
9.   if (x>y)
10.  m = y;
11. else if (x>z)
12.  m = x;
13. print(m);

Branches

DU-pairs

<table>
<thead>
<tr>
<th>stmt.</th>
<th>score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>.71</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>.71</td>
<td>2</td>
</tr>
</tbody>
</table>
Combining Coverage Types

Statements

```c
mid(): int x, y, z, m;
1. read(x, y, z);
2. m = z;
3. if (y<z)
4.   if (x<y)
5.     m = y;
6.   else if (x<z)
7.     m = y;
8.   else
9.     if (x>y)
10.    m = y;
11.   else if (x>z)
12.    m = x;
13. print(m);
```

Branches

DU-pairs

<table>
<thead>
<tr>
<th>stmt.</th>
<th>score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>.71</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>.71</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stmt.</th>
<th>score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.71</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>.71</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>.71</td>
<td>3</td>
</tr>
</tbody>
</table>
Combining Coverage Types

**Statements**

```c
mid(): int x, y, z, m;
1. read(x, y, z);
2. m = z;
3. if (y<z)
   4.   if (x<y)
      5.     m = y;
   6. else if (x<z)
      7.     m = y; // m = x
8. else
9.   if (x>y)
10.  m = y;
11.  else if (x>z)
12.  m = x;
13. print(m);
```

**Branches**

- EN
- T
- F

**DU-pairs**

- .58
- .71

**Tables**

<table>
<thead>
<tr>
<th>stmt.</th>
<th>average score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(.50+.71)/2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stmt.</th>
<th>score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>.71</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>.71</td>
<td>2</td>
</tr>
</tbody>
</table>

- Fault
Combining Coverage Types

Statements

```c
mid(): int x,y,z,m;
1. read(x,y,z);
2. m = z;
3. if (y<z)
   4. if (x<y) m = y;
   5. else if (x<z) m = y;
   6. else m = m;
8. else
   9. if (x>y) m = y;
10. else if (x>z) m = x;
11. else m = x;
12. print(m);
```

Branches

DU-pairs

<table>
<thead>
<tr>
<th>stmt.</th>
<th>average score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.61</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>.65</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>.71</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>.56</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stmt.</th>
<th>score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>.71</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>.71</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>.71</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stmt.</th>
<th>score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.71</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>.71</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>.71</td>
<td>3</td>
</tr>
</tbody>
</table>
Experiment 2

Goal: compare combinations with individual types

Setup

- Three combinations of scores for statements
  - $\text{avg-BD}(s) =$ the average score of branches and du-pairs only, associated with $s$
  - $\text{avg-SBD}(s) =$ the average score of all statements, branches, and du-pairs associated with statement $s$
  - $\text{max-SBD}(s) =$ the maximum score of all statements, branches, and du-pairs associated with statement $s$
- Same as Experiment 1: 14 subjects, 107 faults
  - We measure the difference in cost with the ideal individual type
Experiment 2: Results

Difference in cost w.r.t. the ideal individual type

<table>
<thead>
<tr>
<th>measure</th>
<th>stmt</th>
<th>branch</th>
<th>du-pair</th>
<th>avg-BD</th>
<th>avg-SBD</th>
<th>max-SBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>average diff</td>
<td>5.14%</td>
<td>3.89%</td>
<td>2.68%</td>
<td>1.80%</td>
<td>1.48%</td>
<td>2.77%</td>
</tr>
<tr>
<td>standard dev.</td>
<td>13.28%</td>
<td>11.06%</td>
<td>5.50%</td>
<td>4.66%</td>
<td>4.03%</td>
<td>5.03%</td>
</tr>
</tbody>
</table>

Faults with most different costs (>1% between du-pair and avg-SBD)

avg-SBD is the best combination and better than individual coverage types

ideal of individual types
Experiment 2: Results

Difference in cost w.r.t. the ideal individual type

<table>
<thead>
<tr>
<th>measure</th>
<th>stmt</th>
<th>branch</th>
<th>du-pair</th>
<th>avg-BD</th>
<th>avg-SBD</th>
<th>max-SBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>average diff</td>
<td>5.14%</td>
<td>3.89%</td>
<td>2.68%</td>
<td>1.80%</td>
<td>1.48%</td>
<td>2.77%</td>
</tr>
<tr>
<td>standard dev.</td>
<td>13.28%</td>
<td>11.06%</td>
<td>5.50%</td>
<td>4.66%</td>
<td>4.03%</td>
<td>5.03%</td>
</tr>
</tbody>
</table>

- Combinations reduce difference in cost over the ideal case!
- Combinations require du-pair coverage - can we reduce the overhead of du-pairs?
Monitoring Overhead

Statements

```c
int x, y, z, m;
1. read(x, y, z);
2. m = z;
3. if (y<z)
4.     if (x<y)
5.         m = y;
6.     else if (x<z)
7.         m = x;
8. else
9.     if (x>y)
10.        m = y;
11.     else if (x>z)
12.        m = x;
13. print(m);
```

Branches

- EN
- T
- F

DU-pairs

- 10-20% overhead
- 10-20% overhead
- 60-120% overhead
Inferring DU-pair Coverage

Statements

mid()
    int x, y, z, m;
1.  read(x, y, z);
2.  m = z;
3.  if (y<z)
4.    if (x<y)
5.      m = y;
6.    else if (x<z)
7.      m = x;
8.  else
9.    if (x>y)
10.   m = y;
11.   else if (x>z)
12.    m = x;
13.  print(m);

Branches

DU-pairs

(Santelices & Harrold)
ASE 2007
Inferring DU-pair Coverage

Statements

```c
mid()
    int x,y,z,m;
1.   read(x,y,z);
2.    m = z;
3.    if (y<z)
4.      if (x<y)
5.        m = y;
6.      else if (x<z)
7.        m = x;
8.    else
9.      if (x>y)
10.     m = y;
11.    else if (x>z)
12.     m = x;
13.    print(m);
```

Branches

DU-pairs

(Santelices & Harrold
ASE 2007)
Inferring DU-pair Coverage

Statements

```c
mid()
    int x, y, z, m;
1.  read(x, y, z);
2.  m = z;
3.  if (y<z)
4.    if (x<y)
5.      m = y;
6.    else if (x<z)
7.      m = x;
8.  else
9.    if (x>y)
10.   m = y;
11.   else if (x>z)
12.   m = x;
13. print(m);
```

Branches

DU-pairs

(Santelices & Harrold) ASE 2007
Inferring DU-pair Coverage

**Statements**

```c
mid()
    int x, y, z, m;
1. read(x, y, z);
2. m = z;
3. if (y<z)
4.    if (x<y)
5.      m = y;
6.    else if (x<z)
7.      m = x;
8.  else
9.    if (x>y)
10.     m = y;
11.   else if (x>z)
12.     m = x;
13. print(m);
```

**Branches**

```
6T ⇒ 1→7, 7→m→13
```

**DU-pairs**

(Santelices & Harrold
ASE 2007)
Experiment 3

**Goal:** comparison using inferred du-pair coverage

**Setup**

- Replace du-pair coverage with inferred coverage
  - du-pair-approx(s)
  - avg-BD-approx(s)
  - avg-SBD-approx(s)
  - max-SBD-approx(s)

- Same as Experiments 1 and 2: 14 subjects, 107 faults
  - We measure the difference in cost with the ideal individual type

**Note:** we only monitor for branch coverage now!
### Experiment 3: Results

Difference in cost w.r.t. the **ideal individual type**

<table>
<thead>
<tr>
<th>measure</th>
<th>stmt</th>
<th>branch</th>
<th>du-pair approx</th>
<th>avg-BD approx</th>
<th>avg-SBD approx</th>
<th>max-SBD approx</th>
</tr>
</thead>
<tbody>
<tr>
<td>average diff</td>
<td>5.14%</td>
<td>3.89%</td>
<td>4.26%</td>
<td>2.64%</td>
<td>2.44%</td>
<td>3.71%</td>
</tr>
<tr>
<td>standard dev.</td>
<td>13.28%</td>
<td>11.06%</td>
<td>10.01%</td>
<td>8.02%</td>
<td>7.73%</td>
<td>7.70%</td>
</tr>
</tbody>
</table>

- Faults with most different costs (>1% between *du-pair* and *avg-SBD*)
  - *avg-SBD-approx* is the best combination for inferred *du-pair* coverage

---

**background**

**compare**

**combine**

**infer**

**conclude**
**Experiment 3: Results**

Difference in cost w.r.t. the ideal individual type

<table>
<thead>
<tr>
<th>measure</th>
<th>stmt</th>
<th>branch</th>
<th>du-pair approx</th>
<th>avg-BD approx</th>
<th>avg-SBD approx</th>
<th>max-SBD approx</th>
</tr>
</thead>
<tbody>
<tr>
<td>average diff</td>
<td>5.14%</td>
<td>3.89%</td>
<td>4.26%</td>
<td>2.64%</td>
<td><strong>2.44%</strong></td>
<td>3.71%</td>
</tr>
<tr>
<td>standard dev.</td>
<td>13.28%</td>
<td>11.06%</td>
<td>10.01%</td>
<td>8.02%</td>
<td>7.73%</td>
<td>7.70%</td>
</tr>
</tbody>
</table>

**o** Precise du-pair coverage is more effective than inferred coverage

**o** Combinations retain their advantage over individual types if we use inferred du-pair coverage, with significantly less overhead
Summary

Experiment 1

Different faults are best found using different coverage types
**Summary**

**Experiment 1**

Different faults are best found using different coverage types

But, there is no way to know beforehand which type to choose
Summary

Experiment 1
Different faults are best found using different coverage types

Experiment 2
Combining coverage types gives more effective fault localization than any individual type
Summary

Experiment 1
Different faults are best found using different coverage types

Experiment 2
Combining coverage types gives more effective fault localization than any individual type

Instrumentation for du-pairs incurs a relatively high runtime overhead
Combining coverage types gives more effective fault localization than any individual type. Different faults are best found using different coverage types. Performing du-pair inferencing with only cheap branch instrumentation significantly reduces run-time overhead while retaining better combination effectiveness over individual types.
Future Work

- Compare and combine additional lightweight coverage types (e.g., methods, acyclic paths)
- Experiment with additional heavyweight coverage (e.g., slice fragments) inferred from lighter-weight coverage (e.g., branches, paths)
- Perform more experiments on more subjects and faults, especially different types of faults
Conclusion

- First thorough comparison of effectiveness of different coverage types on fault localization
- First technique that combines coverage types to improve fault-localization effectiveness
- Application of du-pair inference that produces effective results with low overhead