Lokalisieren von Defekten in Software mit Hilfe von Data Mining

Klemens Böhm
A problem has been detected and Windows has been shut down to prevent damage to your computer.

**DRIVER_IRQL_NOT_LESS_OR_EQUAL**

If this is the first time you've seen this stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then Select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0x0000000C, 0x00000002, 0x00000000, 0xF86B5A89)

*** gV3.sys - Address F86B5A89 base at F86B5000, DateStamp 3dd991eb

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further assistance
Locating Bugs in Software

- Software is almost never shipped bug-free, even if tested extensively.
- Debugging is time consuming and expensive.
- Automated localisation would be of great help.
- Idea:
  - Locate bugs with **data mining** techniques.
  - Use an approach based on **weighted graph mining**.
Outline

1. Motivation

2. Call Graphs

3. Defect Localization

4. Dataflow-Enabled Defect Localization

5. Evaluation

6. Conclusions
Call Graphs

- Program executions as call graphs:
  - methods → nodes
  - method calls → edges
Call Graphs

- Program executions as call graphs:
  - methods → nodes
  - method calls → edges
- Bugs in the call graph:
  - **Structure affecting**
    (existing approaches)
    E.g., a bug
    in an if-condition in a

![Call Graph Diagram]

Motivation                          Call Graphs                          Defect Localisation                          Dataflow                          Evaluation                          Conclusions

Klemens Böhm – Lokalisieren von Defekten in Software mit Hilfe von Data Mining

4/25
Call Graphs

- Program executions as call graphs:
  - methods → nodes
  - method calls → edges

- Bugs in the call graph:
  - **Structure affecting** (existing approaches)
    E.g., a bug in an if-condition in a
  - **Call frequency affecting** (new in our contribution)
    E.g., a bug in a loop-condition in c
Reduction of Call Graphs

- Millions of method calls are very common.
- Reduction of call graphs is necessary.

- Weights have not been used in previous reductions.

Example of reduced call graph.
Outline

1. Motivation

2. Call Graphs

3. Defect Localization

4. Dataflow-Enabled Defect Localization

5. Evaluation

6. Conclusions
How to Use Call Graphs?

- Existing idea:
  - Look at **program executions** (correct ones as well as failing ones) represented as **call graphs** and analyse the graph structure (i.e., graphs without weights).
  - Identify patterns typical of failing executions. **Frequent subgraph mining**.
  - In a nutshell, this gives way to finding structure-affecting bugs.

- New aspect:
  Explicitly analyse **call frequencies (edge weights)** besides graph structures.
Weighted Graph Mining

- No algorithm available for **weighted graph mining** (as well as no problem formulation).

- How to make use of edge weights?
  - Preprocessing
    - Discretisation of weights
      (We are currently investigating this.)
  - Postprocessing
    (our approach already studied and presented subsequently)
    - Graph mining without weights
    - Subsequent detailed analysis of weights
Finding Discriminating Edge Weights

(1) Apply **frequent subgraph mining** to **reduced call graphs** of correct and failing program executions (ignore the weights in this step)

(2) Consider only subgraphs occurring in both correct and failing executions

(3) Analyse the edge weights

Example graph found by frequent subgraph mining:

```
  a --> c --> b
```

- Average weight of a → c in correct executions: 1
- Average weight of a → c in failing executions: 1
- Average weight of c → b in correct executions: 3
- Average weight of c → b in failing executions: 30
Finding Discriminating Edge Weights

(1) Apply frequent subgraph mining to reduced call graphs of correct and failing program executions (ignore the weights in this step)

(2) Consider only subgraphs occurring in both correct and failing executions

(3) Analyse the edge weights

Example graph found by frequent subgraph mining:

- Average weight of a→c in correct executions: 1
- Average weight of a→c in failing executions: 1
- **Average weight of c→b in correct executions: 3**
- **Average weight of c→b in failing executions: 30**
Entropy Based Scoring

- Assemble a table which contains every edge in every frequent subgraph:

\[
\begin{array}{c}
\text{SG}_1 \\
a \rightarrow c \rightarrow b \\
\text{SG}_2 \\
a \rightarrow b
\end{array}
\]

<table>
<thead>
<tr>
<th>Execution</th>
<th>SG(_1)</th>
<th>SG(_1)</th>
<th>SG(_2)</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution(_1)</td>
<td>1</td>
<td>30</td>
<td>6</td>
<td>...</td>
</tr>
<tr>
<td>Execution(_2)</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>...</td>
</tr>
</tbody>
</table>

- Result: Ranking of the columns (edges)
## Integration of Structural Evidence

- **Motivation**
- **Call Graphs**
- **Defect Localisation**
- **Dataflow**
- **Evaluation**
- **Conclusions**

### Reduced Call Graphs

- Frequent subgraph mining (not considering weights)
  - In failing and correct
  - Entropy based scoring (based on edge weights)
  - Structural scoring (based on support)

### Combination

- Combined method ranking
Integration of Structural Evidence

- reduced call graphs
- frequent subgraph mining (not considering weights)
  - entropy based scoring (based on edge weights)
  - structural scoring (based on support)
- combination
  - combined method ranking

Motivation

Call Graphs

Defect Localisation

Dataflow

Evaluation

Conclusions
Integration of Structural Evidence

- Usage of two kinds of evidence: **frequency** (left) and **structure** (right)
- Both types of bugs can be located: **call frequency** and **structure affecting ones**

Usage of two kinds of evidence:
- **frequency** (left) and **structure** (right)
- Both types of bugs can be located: **call frequency** and **structure affecting ones**

- Motivation
- Call Graphs
- Defect Localisation
- Dataflow
- Evaluation
- Conclusions
## Results

**Example output:**

<table>
<thead>
<tr>
<th>METHOD</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputscan()</td>
<td>0.9833</td>
</tr>
<tr>
<td>showinsert()</td>
<td>0.9204</td>
</tr>
<tr>
<td>showdelete()</td>
<td>0.4876</td>
</tr>
<tr>
<td>oldconsume()</td>
<td>0.4876</td>
</tr>
<tr>
<td>addsymbol()</td>
<td>0.2428</td>
</tr>
</tbody>
</table>
Results

Example output:

<table>
<thead>
<tr>
<th>METHOD</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputscan()</td>
<td>0.9833</td>
</tr>
<tr>
<td>showinsert()</td>
<td>0.9204</td>
</tr>
<tr>
<td>showdelete()</td>
<td>0.4876</td>
</tr>
<tr>
<td>oldconsume()</td>
<td>0.4876</td>
</tr>
<tr>
<td>addsymbol()</td>
<td>0.2428</td>
</tr>
</tbody>
</table>

- The bug has been instrumented in showinsert().
- A software developer has to check two methods only.
- We do not try to explain/fix bugs, just to locate them!
Outline

1. Motivation
2. Call Graphs
3. Defect Localization
4. Dataflow-Enabled Defect Localization
5. Evaluation
6. Conclusions
Challenge: Dataflow-Affecting Bugs

- Not all defects affect the call graph.
- Example (failing execution):

1. Method \( \texttt{b} \) calls \( \texttt{c}(12, 33) \).
2. Method \( \texttt{c} \) calculates a wrong return value 8.
3. Method \( \texttt{b} \) prints value 8.

Graph structure and weights remain unaffected.
\( \Rightarrow \) Call-graph mining cannot find the defect.

- Approach: Extend call graphs with dataflow information and analyse these extended graphs.
Example dataflows (parameter and return values) of \( \text{int } c(\text{int } p1, \text{int } p2) \) from \( b \):

<table>
<thead>
<tr>
<th>Exec.</th>
<th>( p1 )</th>
<th>( p2 )</th>
<th>( r )</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>43</td>
<td>12</td>
<td>correct</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>44</td>
<td>11</td>
<td>correct</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>correct</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>33</td>
<td>8</td>
<td>failing</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td>27</td>
<td>6</td>
<td>failing</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>28</td>
<td>5</td>
<td>failing</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>23</td>
<td>7</td>
<td>failing</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>2</td>
<td>10</td>
<td>correct</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>47</td>
<td>13</td>
<td>correct</td>
</tr>
</tbody>
</table>

In real executions, one method can be called million times.
To deal with this data, we need to aggregate it.
Our Approach: Discretisation with CAIM

- **CAIM** (class-attribute interdependence maximization)
- **class-aware** (correct, failing)
- **parameter-free**, i.e., it automatically determines a possibly small number of intervals, homogeneous wrt. the class

(a) Exemplar call data.

<table>
<thead>
<tr>
<th>Exec.</th>
<th>p1</th>
<th>p2</th>
<th>r</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>43</td>
<td>12</td>
<td>correct</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>44</td>
<td>11</td>
<td>correct</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>correct</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>33</td>
<td>8</td>
<td>failing</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td>27</td>
<td>6</td>
<td>failing</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>28</td>
<td>5</td>
<td>failing</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>23</td>
<td>7</td>
<td>failing</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>2</td>
<td>10</td>
<td>correct</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>47</td>
<td>13</td>
<td>correct</td>
</tr>
</tbody>
</table>

(b) Intervals generated.

<table>
<thead>
<tr>
<th>Value</th>
<th>Intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>i1: [1; 11.5]</td>
</tr>
<tr>
<td></td>
<td>i2: (11.5; 23]</td>
</tr>
<tr>
<td>p2</td>
<td>i1: [2; 13.5]</td>
</tr>
<tr>
<td></td>
<td>i2: (13.5; 38]</td>
</tr>
<tr>
<td></td>
<td>i3: (38; 47]</td>
</tr>
<tr>
<td>r</td>
<td>i1: [5; 8.5]</td>
</tr>
<tr>
<td></td>
<td>i2: (8.5; 13]</td>
</tr>
</tbody>
</table>

(c) Discretised data.

<table>
<thead>
<tr>
<th>Exec.</th>
<th>p1</th>
<th>p2</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i1</td>
<td>i3</td>
<td>i2</td>
</tr>
<tr>
<td>1</td>
<td>i1</td>
<td>i3</td>
<td>i2</td>
</tr>
<tr>
<td>2</td>
<td>i2</td>
<td>i2</td>
<td>i1</td>
</tr>
<tr>
<td>3</td>
<td>i2</td>
<td>i2</td>
<td>i1</td>
</tr>
<tr>
<td>3</td>
<td>i2</td>
<td>i2</td>
<td>i1</td>
</tr>
<tr>
<td>4</td>
<td>i1</td>
<td>i1</td>
<td>i2</td>
</tr>
<tr>
<td>4</td>
<td>i1</td>
<td>i3</td>
<td>i2</td>
</tr>
</tbody>
</table>
Dataflow-Enabled Call Graphs (DEC Graphs)

- How to represent discretised dataflow data in call graphs?
- Attach **tuples of weights** (i.e., counters) to edges:
  
  \[(t; p_{i1}; \ldots; p_{in1}; \ldots; p_{im}; \ldots; p_{inm}; r_{i1}; \ldots; r_{inr})\]

- **t**: total number of calls (as before)
- **p_{1}; \ldots; p_{m}; r**: parameter values and return value
- **i_{1}; \ldots; i_{nx}**: intervals of the parameter/return values

**Example call** \(b \rightarrow c\):

\[(t; p_{i1}; p_{i2}; p_{i1}; p_{i2}; p_{i3}; r_{i1}; r_{i2})\]
Defect Localisation with DEC Graphs

1. Assemble classified DEC graphs for all executions.
2. Perform frequent subgraph mining (CloseGraph). We use subgraphs as contexts to detect defects that occur in certain situations only.
3. Assemble a feature table with all tuple elements from all edges in all frequent subgraphs separately:

<table>
<thead>
<tr>
<th>Exec.</th>
<th>$sg_1$</th>
<th>$sg_2$</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>main → b</td>
<td>b → c</td>
<td>main → a</td>
</tr>
<tr>
<td>main</td>
<td>t</td>
<td>$p_1^1/t$</td>
<td>$p_2^1/t$</td>
</tr>
<tr>
<td>$g_1$</td>
<td>2 3</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$g_2$</td>
<td>4 1</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
</tbody>
</table>

4. Derive a ranking of the columns (and thus the methods) with entropy-based feature selection (GainRatio).

Various optimisations.
Outline

1. Motivation

2. Call Graphs

3. Defect Localization

4. Dataflow-Enabled Defect Localization

5. Evaluation

6. Conclusions
Evaluation of Defect Localisation with DEC Graphs

- A data-centric application: Weka (class DecisionStump)
- 16 instrumented defects, similar to related evaluations. Most dataflow-affecting bugs indirectly affect the call graph.
- 90 executions of every version with UCI data
- The result table contains the ranking position, i.e., number of methods one has to inspect to find a defect:

| Experiment \ Defect | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | Ø |
|---------------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| DEC graphs          | 3 | 3 | 1 | 3 | 2 | 2 | 12| 3 | 1 | 1  | 2  | 1  | 2  | 1  | 2  | 1  | 3  | 2.3|
| Non-DEC graphs      | 1 | 1 | 11| 13| 10| 3 | 13| 10| 9 | 6  | 3  | 8  | 1  | 3  | 8  | 10 | 6.1|

- With improvements only 1.5 methods on average (out of 30 actually executed methods).
Conclusions

A domain-specific data-mining problem: software-defect localisation

Contributions:

- A domain and problem-specific data representation: DEC graphs (discretisation)
- An analysis process for DEC graphs (subgraph mining, feature selection)
- Localization of dataflow-affecting bugs besides other bug classes

Future improvements:

- Dataflows through global variables
- Scalable analysis of large software projects
- Parallel programs
Acknowledgements

Thanks to Frank Eichinger for letting me use his slides.