Improved Software Fault Detection with Graph Mining

Frank Eichinger Klemens Böhm Matthias Huber

Institute for Program Structures and Data Organisation (IPD)
Universität Karlsruhe (TH), Germany

5th July 2008

Locating Bugs in Software

- Software is almost never shipped bug-free, even if tested extensively.
- Particularly challenging are:
  - **Noncrashing bugs** – no stack trace available
  - **Occasional bugs** – occur just with some input data
- Some resources are available, but software projects are way too large for a complete review.

Locating Bugs in Software

- Software is almost never shipped bug-free, even if tested extensively.
- Particularly challenging are:
  - **Noncrashing bugs** – no stack trace available
  - **Occasional bugs** – occur just with some input data
- Some resources are available, but software projects are way too large for a complete review.

- Idea:
  - Locate **noncrashing occasional bugs** with **data mining** techniques.

Locating Bugs in Software

- Software is almost never shipped bug-free, even if tested extensively.
- Particularly challenging are:
  - **Noncrashing bugs** – no stack trace available
  - **Occasional bugs** – occur just with some input data
- Some resources are available, but software projects are way too large for a complete review.

- Idea:
  - Locate **noncrashing occasional bugs** with **data mining** techniques.
  - Using a **weighted graph mining** approach

Outline

Data Mining in Software Engineering

- Traditional data mining techniques process feature vectors of numerical and categorical data.
- In software engineering, this can be
  - different code metrics (static analysis)
  - data gained from instrumentation (dynamic analysis)

	METRIC 1	METRIC 2	METRIC 3	...
Software artefact 1	123	5	12	...
Software artefact 2	222	8	12	...
...	...	...	...	...

- Searched are patterns or properties which are more likely in buggy software.

Challenges with Software Metrics

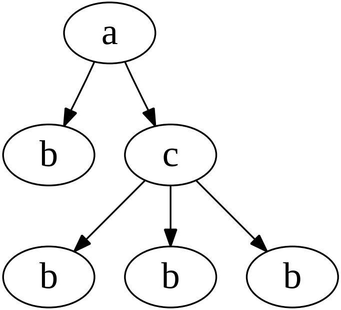
- Shortcomings of relational metric collections:
  - Static analysis
    - Even hundreds of metrics
      describe a program insufficiently.
    - Static code metrics help little for locating bugs.
  - Dynamic analysis (instrumentation)
    - Tradeoff: runtime vs. amount of relevant data

Challenges with Software Metrics

- Shortcomings of relational metric collections:
  - Static analysis
    - Even hundreds of metrics describe a program insufficiently.
    - Static code metrics help little for locating bugs.
  - Dynamic analysis (instrumentation)
    - Tradeoff: runtime vs. amount of relevant data
- Idea:
  - Look at **program executions** represented as **call graph** and analyse its structure.
  - Identify substructures typical for failing executions.
    - Requires a test oracle, which is typically available.
  - Analyse the **call frequencies** (**edge weights**) as well!

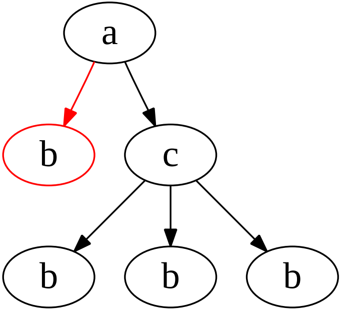
Call Graphs

- Call graphs are rooted ordered trees.
- Program executions as call graphs:
  - Methods → nodes
  - Method calls → edges
- Bugs in the call tree:



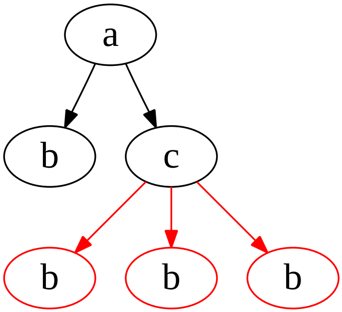
Call Graphs

- Call graphs are rooted ordered trees.
- Program executions as call graphs:
  - Methods → nodes
  - Method calls → edges
- Bugs in the call tree:
  - **Structure affecting**
    - E.g., a bug in an `if`-condition in `a`

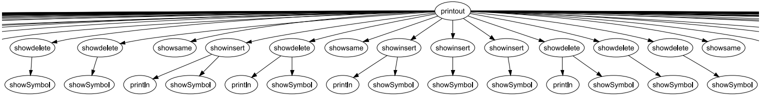


Call Graphs

- Call graphs are rooted ordered trees.
- Program executions as call graphs:
  - Methods → nodes
  - Method calls → edges
- Bugs in the call tree:
  - **Structure affecting**
    - E.g., a bug in an `if`-condition in `a`
  - **Call frequency affecting**
    - E.g., a bug in a loop-condition in `c`

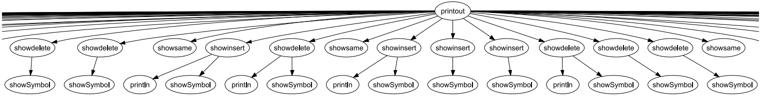


Reduction of Call Graphs

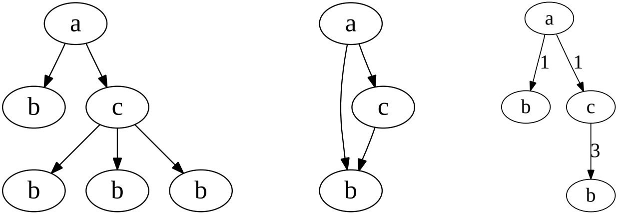


- Millions of method calls are very common!

Reduction of Call Graphs



- Millions of method calls are very common!
- Several reduction techniques exist, e.g.:

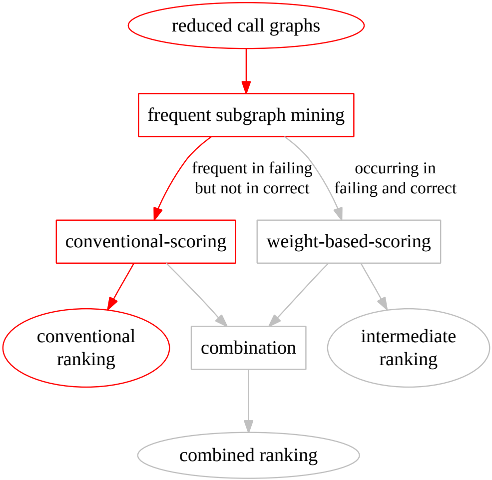


Conventional Approach (di Fatta et al., 2006)

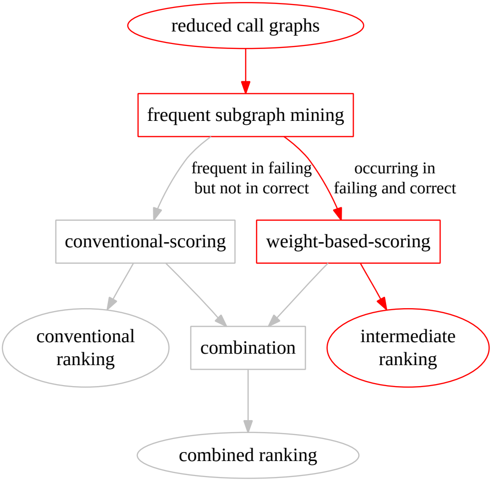
1. Input data:
   - Collection of call graphs classified as **correct** or **failing**.
2. Search for frequent subgraphs with an arbitrary algorithm.
3. Identify discriminative subgraphs: frequent within the **faulty** but not within the **correct** executions.
   - The methods within these subgraphs display an increased likelihood of containing a bug.

→ Here, only structural differences are considered – no call frequencies of graphs occurring in both sets.

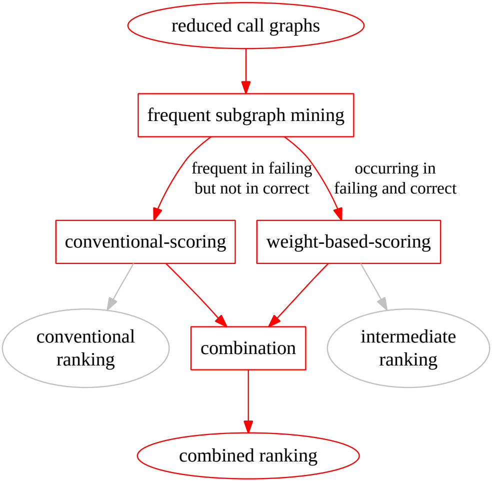
Framework for Locating Bugs



Framework for Locating Bugs



Framework for Locating Bugs



- Both types of bugs are considered: structure and call frequency affecting ones
- Integration of the traditional approach

Entropy-based Ranking of Edge Weights

- Identification of edge weights which most differentiate between the two classes
- Every edge in every frequent subgraph is considered:

	SG_1 $a \to b$	SG_1 $a \to c$	SG_2 $a \to b$	$\cdots$	$Class$
$Graph_1$	2	1	6	$\cdots$	$failing$
$Graph_2$	0	0	4	$\cdots$	$correct$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

- Application of an entropy-based feature selection algorithm to the table.
- Result: Ranking of the columns (edges)

Results

- Example output:

	METHOD	SCORE
1	`inputscan()`	0.9833
2	`showinsert()`	0.9204
3	`showdelete()`	0.4876
4	`oldconsume()`	0.4876
5	`addSymbol()`	0.2428

Results

- Example output:

	METHOD	SCORE
1	`inputscan()`	0.9833
2	**`showinsert()`**	**0.9204**
3	`showdelete()`	0.4876
4	`oldconsume()`	0.4876
5	`addSymbol()`	0.2428

- The bug was instrumented in `showinsert()`.
- A software developer has to check two methods only.
- Low line numbers are better!

Experiments

- Setting:
  - Open source Java tool, 25 methods
  - 9 artificial but realistic bugs
  - 100 program executions each
- Experimental results:
  - Displayed is the number of methods to be reviewed.

Exp. \ Bug No.	1	2	3	4	5	6	7	8	9
Conventional	3	–	1	4	6	4	3	3	1
Intermediate	3	3	1	1	1	3	3	1	–
Combined	1	3	1	2	2	1	2	1	3

- Localisation precision increased by 2.4.

Conclusion

- Summary of contributions
  - New call graph reduction variant
  - Mining of weighted graphs in a classification scenario
    - Combining structural and numerical techniques
    - Applicable in other domains
  - Considering weights of non-discriminative patterns
- Results in software engineering
  - Ability to detect a new important class of bugs
  - Doubled precision of bug localisations
- Current and future work
  - Weight based constraints
  - Mining of large graphs/large software projects
  - Other fields of application

Questions?

Thank you for your attention!