Search Based Software Engineering: Foundations, Challenges and Recent Advances

Marouane Kessentini
marouane@umich.edu

SBSE Research Lab, CIS Department,
College of Engineering and Computer Science,
University of Michigan, Dearborn, USA

IEEE WCCI 2016 | 2016 IEEE World Congress on Computational Intelligence.
Acknowledgments

• Many thanks to Prof. Mark Harman (Founder of Search-Based Software Engineering) for the help to prepare part of this tutorial from the following source:

  – Mark Harman, UCL, UK
Outline

• Philosophical Basis: Science and Engineering

• What is SBSE?

• Recent Advances
  – Bi-Level SBSE for Design Defects Detection
  – Interactive Multi-Objective SBSE for Refactoring
  – Many-Objective SBSE for Software Remodularization

• Challenges and Future Research Directions
• Philosophical Basis: Science and Engineering

• What is SBSE?

• Recent Advances
  – Bi-Level SBSE for Design Defects Detection
  – Interactive Multi-Objective SBSE for Refactoring
  – Many-Objective SBSE for Software Remodularization

• Challenges and Future Research Directions
Outline

• Philosophical Basis: Science and Engineering

• What is SBSE?

• Recent Advances
  – Bi-Level SBSE for Design Defects Detection
  – Interactive Multi-Objective SBSE for Refactoring
  – Many-Objective SBSE for Software Remodularization

• Challenges and Future Research Directions
<table>
<thead>
<tr>
<th>Scientist:</th>
<th>Engineer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is true</td>
<td>What is possible</td>
</tr>
<tr>
<td>Correctness</td>
<td>Within tolerance</td>
</tr>
<tr>
<td>Model the world to understand</td>
<td>Model the world to manipulate</td>
</tr>
</tbody>
</table>
Computer scientist:
What is true about computation
Proof correctness
Make it perfect

Software engineer:
What is possible with software
Test for imperfection
find where to improve
prove correctness
make it perfect

where possible ...

... and where impossible ...

test for imperfection
find where to improve
Engineering Words

tolerance
With acceptable bounds

optimise
Improve performance

Reduce cost

Optimize
Within constraints

Optimization: so good they named it twice!

First in English …
Then in American
What is SBSE?

- In SBSE we apply search techniques to search large search spaces, guided by a fitness function that captures properties of the acceptable software artefacts we seek.
  
  like google search?

  like code search?

  like breadth first search?
What is SBSE?

• In SBSE we apply search techniques to search large search spaces, guided by a fitness function that captures properties of the acceptable software artefacts we seek.

  Genetic Programming
  Ant Colonies
  Hill Climbing
  Harmony Search
  Tabu Search
  Particle Swarm Optimization
  Simulated Annealing
What is SBSE?

Search-Based Optimization

Software Engineering
• Search Based Software Engineering
  – Write a method to determine which is the better of two solutions

• Conventional Software Engineering
  – Write a method to construct a perfect solution
• Search Based Software Engineering
  – Write a **fitness function** to guide automated search

• Conventional Software Engineering
  – Write a method to construct a perfect solution
...but...

why is Software Engineering different?
In situ fitness test

Physical Engineering

Cost: 20,000$

Virtual Engineering

Cost: 0 $
Spot the Difference

**Traditional Engineering Artifact**

- **Optimization goals**
  - Maximize compression
  - Minimize fuel consumption

- **Fitness computed on a representation**

**Traditional Engineering Artifact**

- **Optimization goals**
  - Maximize cohesion
  - Minimize coupling

- **Fitness computed on a representation**
...but...

why is SBSE growing very fast?
let’s listen to software engineers ...

... what sort of things do they say?
Software Engineers Say…

Requirements: We need to satisfy business and technical concerns

Management: We need to reduce risk while maintaining completion time

Design: We need increased cohesion and decreased coupling

Testing: We need fewer tests that find more nasty bugs

Refactoring: We need to optimize for all metrics M1,..., Mn

All have been addressed in the SBSE literature
Software Engineers Say…

- Capture requirements
- Generate tests
- Model Transformation
- Refactoring

Minimize
- Cost
- Development time

Maximize
- Satisfaction
- Fairness
Software Engineers Say...

- Capture requirements
- Generate tests
- Model Transformation
- Refactoring

Minimize
- Number of tests
- Execution time

Maximize
- Code coverage
- Fault coverage
Software Engineers Say…

- Capture requirements
- Generate tests
- Model Transformation
- Refactoring

Minimize

- Rules correctness

Maximize

- Rules complexity
- Models quality
Software Engineers Say...

- Capture requirements
- Generate tests
- Model Transformation
- Refactoring

Minimize

- Number of refactorings

Maximize

- Quality factors
- Semantics preservation
The Advantages of SBSE

- Scalable
- Generic
- Robust
- Realistic
Software Engineering

SBSE is so generic...

Search Based Software Engineering

Optimization Techniques

Solution representation

encoding

Fitness Function

Function defined to evaluate solutions

Change operator

Search Problem

Search Problem

Software Engineering Problem
SBSE is so generic...

Selection Problems

Requirements Assignment

- Select
- Requirements: $R_1, R_2, \ldots, R_n$
- Customers: $C_1, C_2, \ldots, C_m$

Regression Testing

- Select
- Test Cases: $T_1, T_2, \ldots, T_n$
- Covered Items in Code (e.g. statements)

Prioritization Problems

- Requirements: $R_1, R_2, \ldots, R_n$ (Permutation) $\rightarrow$ Optimal Order of Development: $R_4, R_7, R_{12}, \ldots$
- Test Cases: $T_1, T_2, \ldots, T_n$ (Permutation) $\rightarrow$ Optimal Order of Re-testing: $T_3, T_5, T_2, \ldots$
Requirements and regression testing: Really different?

Alone
All one
Our Recent Advances in SBSE

- Bi-Level Optimization
- Dynamic Interactive Multi-Objective Optimization
- Many-Objective Optimization

Design Defects Detection
- (TOSEM, 2015)

Design Defects Correction (Refactoring)
- (ASE, 2015)

Remodularization
- (TOSEM, 2015)
Our Recent Advances in SBSE

Design Defects Detection

Bi-Level Optimization

Interactive Multi-Objective Optimization

Many-Objective Optimization

(TOSEM, 2015)

Re-modularization
Software Refactoring

• Software changes frequently
  • Add new functionalities
  • Correcting bugs
  • Adaptation to environment changes
  • Software engineers spend 60% of their time in understanding the code

• Easiness to accommodate changes depends on the software quality
  • Refactoring
Software Refactoring

• Refactoring
  – The process of improving a code after it has been written by changing its internal structure without changing the external behavior (Fowler et al., '99)
  – Examples: *Move method, extract class, move attribute, ...*

• Two main refactoring steps
  1. detection of code fragments to improve (e.g., design defects)
  2. identification of refactoring solutions
Step 1: Design defects detection

- **Design defect** introduced during the initial design or during evolution
  - Anomalies, anti-patterns, bad smells...
  - Design situations that adversely affect the development of a software (not bugs)
  - Examples: *Blob, spaghetti code, functional decomposition, …*
The Blob Example

• Definition
  – Procedural-style design leads to one object with numerous responsibilities and most other objects only holding data or executing simple processes.

• Symptoms
  – A Blob is a controller class, abnormally large, with almost no parents and no children. It mainly uses data classes, i.e. very small classes with almost no parents and no children (Brown et al. ’98).
Step 2: Refactoring

Refactoring

Move method
Extract class
Move field
Add association

Blob
Existing Work

- Design defects detection
  - Manual (Brown et al. '98, Fowler and Beck '99)
  - Metrics-based (Marinescu et al. '04, Salehie et al. '06, Maiga et al. '12)
  - Visual (Dhambri et al. '08, Langelier et al. '05)
  - Symptoms-based (Moha et al. '08, Murno et al. '08)

Definition ➔ symptoms ➔ detection algorithm
Existing Work

• Detection issues
  – No consensual definition of symptoms
  – The same symptom could be associated to many defect types
  – Difficulty to automate symptom’s evaluation
  – Require an expert to manually write and validate detection rules
Limitations ...

- Detection rules heavily depend on the base of examples (coverage, quality, etc.)

- To generate good detection rules, we need to have examples from similar contexts
Problem

• Large exhaustive list of quality metrics
• Large number of possible threshold values

Search problem to explore this huge space
**Bi-Level Code-Smells Detection**

**Upper-Level:** Detection rules generation using Genetic Programming
- Objective: Maximize the coverage of code-smell examples and artificial code-smells generated by the lower level

**Lower-Level:** Generation of artificial code-smells
- Objective: Maximize the distance with well-designed code-examples and maximize the number of artificial code-smells that are not detected by the leader (detection rules)
Bi-Level Optimization

Upper level (Leader)

Lower level (Follower)
Solution Representation

- **Tree:**
  - Leaf node (Terminal): metrics and their thresholds
  - Internal node (Functions): logic operators (AND;OR)

- **Vector:**
  - A set of detectors
  - A detector is composed of 7 metrics

- Genetic Programming for detection rules generation
- Genetic Algorithm for detectors generation
- Code-smells examples
- Quality metrics
- Well designed code examples
- Artificial code-smells examples
- Tree: Leaf node (Terminal): metrics and their thresholds
- Tree: Internal node (Functions): logic operators (AND;OR)
- Vector: A set of detectors
- Vector: A detector is composed of 7 metrics
Fitness Functions

- Code-smells examples
  - Genetic Programming for detection rules generation
  - Detection rules

- Quality metrics
  - Evaluating detection-rules solutions:
    - Maximize the coverage of the base of code-smell examples
    - Maximize the number of covered “artificial” code-smells generated by the lower level solutions

- Genetic Algorithm for detectors generation
  - Well designed code examples
  - Artificial code-smells examples

- Quality metrics
  - Evaluating the artificial code-smells examples:
    - Maximize the dissimilarity score between generated code-smells and reference code
    - Maximize the number of generated code-smell examples un-covered by the upper level solutions
Validation: Research Questions

• **RQ1:** How does BLOP perform to detect different types of code-smells (Precision and Recall)?

• **RQ2:** How do BLOP perform compared to existing search-based code-smells detection algorithms?

• **RQ3:** How does BLOP perform compared to the existing code-smells detection approaches not based on the use of metaheuristic search?

• **RQ4:** How does our bi-level formulation scale?
Validation:
Studied Systems

<table>
<thead>
<tr>
<th>Systems</th>
<th>Release</th>
<th>#Classes</th>
<th>#Smells</th>
<th>KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart</td>
<td>v1.0.9</td>
<td>521</td>
<td>82</td>
<td>170</td>
</tr>
<tr>
<td>GanttProject</td>
<td>v1.10.2</td>
<td>245</td>
<td>67</td>
<td>41</td>
</tr>
<tr>
<td>ApacheAnt</td>
<td>v1.5.2</td>
<td>1024</td>
<td>163</td>
<td>255</td>
</tr>
<tr>
<td>ApacheAnt</td>
<td>v1.7.0</td>
<td>1839</td>
<td>159</td>
<td>327</td>
</tr>
<tr>
<td>Nutch</td>
<td>v1.1</td>
<td>207</td>
<td>72</td>
<td>39</td>
</tr>
<tr>
<td>Log4J</td>
<td>v1.2.1</td>
<td>189</td>
<td>64</td>
<td>31</td>
</tr>
<tr>
<td>Lucene</td>
<td>v1.4.3</td>
<td>154</td>
<td>37</td>
<td>33</td>
</tr>
<tr>
<td>Xerces-J</td>
<td>V2.7.0</td>
<td>991</td>
<td>106</td>
<td>238</td>
</tr>
<tr>
<td>Rhino</td>
<td>v1.7R1</td>
<td>305</td>
<td>78</td>
<td>57</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------</td>
<td>--------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>89%</td>
<td>78%</td>
<td>84%</td>
<td>26%</td>
</tr>
<tr>
<td></td>
<td>(77/86)</td>
<td>(71/92)</td>
<td>(74/89)</td>
<td>(34/129)</td>
</tr>
<tr>
<td>GanttProject</td>
<td>88%</td>
<td>80%</td>
<td>82%</td>
<td>28%</td>
</tr>
<tr>
<td>ApacheAnt v1.5.2</td>
<td>90%</td>
<td>84%</td>
<td>86%</td>
<td>26%</td>
</tr>
<tr>
<td>ApacheAnt v 1.7.0</td>
<td>91%</td>
<td>82%</td>
<td>85%</td>
<td>28%</td>
</tr>
<tr>
<td></td>
<td>(149/164)</td>
<td>(142/173)</td>
<td>(144/169)</td>
<td>(54/184)</td>
</tr>
<tr>
<td>Nutch</td>
<td>89%</td>
<td>73%</td>
<td>76%</td>
<td>34%</td>
</tr>
<tr>
<td></td>
<td>(67/76)</td>
<td>(64/88)</td>
<td>(65/86)</td>
<td>(37/131)</td>
</tr>
<tr>
<td>Log4J</td>
<td>89%</td>
<td>71%</td>
<td>77%</td>
<td>32%</td>
</tr>
<tr>
<td></td>
<td>(59/67)</td>
<td>(52/74)</td>
<td>(54/71)</td>
<td>(34/127)</td>
</tr>
<tr>
<td>Lucene</td>
<td>91%</td>
<td>70%</td>
<td>79%</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td>(35/39)</td>
<td>(31/42)</td>
<td>(33/42)</td>
<td>(11/88)</td>
</tr>
<tr>
<td>Xerces-J</td>
<td>91%</td>
<td>75%</td>
<td>80%</td>
<td>17%</td>
</tr>
<tr>
<td>Rhino</td>
<td>90%</td>
<td>74%</td>
<td>79%</td>
<td>14%</td>
</tr>
<tr>
<td></td>
<td>(75/84)</td>
<td>(69/93)</td>
<td>(71/89)</td>
<td>(23/167)</td>
</tr>
</tbody>
</table>
Results

Box plots on three different systems of: (a) precision values, and (b) recall values
Number of Defect Examples

The graph shows the precision of three different methods (BLOP, GP, Co-Evol) as a function of the number of code smell examples. The y-axis represents precision ranging from 0 to 100, while the x-axis represents the number of code smell examples ranging from 0 to 200. The graph indicates that as the number of code smell examples increases, the precision of all methods also increases, with BLOP generally leading in precision.
Comparison with non-search based approach
Industrial Case Study: Ford Motor Company

- 8 software engineers from Ford evaluated the detected defects on the JDI System

<table>
<thead>
<tr>
<th>Systems</th>
<th>Release</th>
<th>#Classes</th>
<th>#Smells</th>
<th>KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDI-Ford</td>
<td>v5.8</td>
<td>638</td>
<td>88</td>
<td>247</td>
</tr>
</tbody>
</table>
Quality gain on JDI-Ford

Quality attributes

Quality gain

- Reusability
- Flexibility
- Understandability
- Effectiveness
Code Smells Relevance

Code smells relevance: JDI-Ford

- Not at all relevant
- Slightly relevant
- Moderately relevant
- Extremely relevant
Usefulness of detected code smells on JDI-Ford

- Refactoring guidance
- Quality Assurance
- Code inspection
- Effort prediction
- Bug prediction
Our Recent Advances in SBSE

Design Defects Detection

Design Defects Correction (Refactoring)

Re-modularization

Bi-Level Optimization

Dynamic Interactive Multi-Objective Optimization

Many-Objective Optimization

(ASE, 2015)
Existing Work

- Metric-based approaches
  - Search-based techniques
    - Find the best sequence of refactorings (Harman et al. ‘07, O’Keeffe et al. ‘08)
  - Analytic approaches
    - Study of relations between some quality metrics and refactoring changes (Sahraoui et al. ‘00, Du Bois et al. ‘04, Moha et al. ‘08)
- Graph-based approaches
  - Graph transformation
    - Software is represented as a graph
    - Refactorings activities as graph production rules (Kataoka et al., ‘01, Heckel et al. ‘95)
Manual refactoring is
• error-prone,
• time consuming,
• not scalable
• not useful for radical refactoring (extensive application of refactorings to correct unhealthy code.)

Fully-automated refactoring
• lacks flexibility (developers have to accept the entire refactoring solution),
• fails to consider developer perspective and feed-back,
• proposes a long static list of refactorings to be applied but developers do not have enough time to apply all of them
DINAR: Dynamic Interactive Multi-objective refactoring

**Online Phase**

- Ranked list of Refactorings
- Dynamic update of refactorings ranking
- Developers action (Accept/Modify/Reject refactoring)

**Offline Phase**

- Interactive NSGA-II to generate refactoring solutions:
  - Objective 1: Improve the quality
  - Objective 2: Reduce the number of refactorings
- New search constraints: List of accepted, modified and rejected refactorings.
- Pareto front of best refactoring solutions
- Identifying the most common features between refactoring solutions to rank them
- A list of ranked refactorings from the best solution
The Three Components of DINAR

1. Upfront generation of refactoring solutions using NSGA-II
NSGA-II: Non-dominated Sorting Genetic Algorithm (K. Deb et al., '02)
• Individual = Refactoring solution
• Sequence of refactoring operations

<table>
<thead>
<tr>
<th>RO</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO1</td>
<td>moveMethod</td>
</tr>
<tr>
<td>RO2</td>
<td>pullUpAttribute</td>
</tr>
<tr>
<td>RO3</td>
<td>extractClass</td>
</tr>
<tr>
<td>RO4</td>
<td>inlineClass</td>
</tr>
<tr>
<td>RO5</td>
<td>extractSuperClass</td>
</tr>
<tr>
<td>RO6</td>
<td>inlineMethod</td>
</tr>
<tr>
<td>RO7</td>
<td>extractClass</td>
</tr>
<tr>
<td>RO8</td>
<td>moveMethod</td>
</tr>
</tbody>
</table>
- Specify the controlling parameters

<table>
<thead>
<tr>
<th>Refactorings</th>
<th>Controlling parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>move method</td>
<td>(sourceClass, targetClass, method)</td>
</tr>
<tr>
<td>move field</td>
<td>(sourceClass, targetClass, field)</td>
</tr>
<tr>
<td>pull up field</td>
<td>(sourceClass, targetClass, field)</td>
</tr>
<tr>
<td>pull up method</td>
<td>(sourceClass, targetClass, method)</td>
</tr>
<tr>
<td>push down field</td>
<td>(sourceClass, targetClass, field)</td>
</tr>
<tr>
<td>push down method</td>
<td>(sourceClass, targetClass, method)</td>
</tr>
<tr>
<td>inline class</td>
<td>(sourceClass, targetClass)</td>
</tr>
<tr>
<td>extract class</td>
<td>(sourceClass, newClass)</td>
</tr>
</tbody>
</table>
Population: set of refactoring solutions
2.a Ranking of refactoring solutions

- Counts the number of occurrence of the refactoring operation $R_{i,j}$ among all the Pareto front solutions.

- Search for common principles among the refactoring solutions.

$$\text{Rank}(R_{i,j}) = \frac{\text{number}_{-}\text{occurrence}}{\text{max}_{-}\text{number}_{-}\text{occurrence}} + \frac{\sum_{k=1}^{i} \text{Sim}(R_{k,j}, \text{recommended}_{-}\text{ref})}{\#\text{recommended}_{-}\text{ref}}$$

- The ranking of refactorings is updated automatically after every feed-back from the developer.
The Three Components of DINAR

2.b Interactive recommendation of refactorings

The list of ranked Refactorings recommended by DINAR
2.b Interactive recommendation of refactorings

The user can modify the suggested refactoring
3. Dynamic update of recommended refactorings

- The input of this component is the new system after major changes are performed by developers on the original one and the latest set of good refactoring solutions.

- The output is a new updated set of non-dominated refactoring solutions that are adapted to the new system using an indicator-based local search.
Results: Manual Correctness

Median manual correctness (MC) value over 31 runs on all the five systems using the different refactoring techniques with a 99% confidence level (α < 1%).
Results: Refactoring Recommendation

Median value of refactorings (PRT) and code elements selected from the top5 on all the five systems.
Results: Refactoring Recommendation

Average time $T$ (minutes) required by developers to finalize a refactoring session.
Results: Feed-back from Developers

– The participants agreed that
  • the interactive dynamic refactoring recommendations are a desirable feature in IDEs
  • the interactive manner of recommending refactorings by DINAR is a useful and flexible way to refactor systems comparing to fully-automated or manual refactorings

– The participants found DINAR helpful
  • for both floss refactoring and root canal refactoring
  • to modify the source code (to add new functionality) while doing refactoring
  • to take the advantages of using multi-objective optimization for software refactoring without the explicit exploration of the set of non-dominated solutions
Our Recent Advances in SBSE

Design Defects Detection
Design Defects Correction (Refactoring)
Re-modularization

Bi-Level Optimization
Dynamic Interactive Multi-Objective Optimization
Many-Objective Optimization

(TOSEM, 2015)
Motivating Example
Motivating Example

Xerces v 2.5.1:
Move class (MutationEventImpl, org.apache.xerces.dom, org.apache.xerces.dom.events)
Xerces v 2.5.1:
Move class (MutationEventImpl, org.apache.xerces.dom, org.apache.xerces.dom.events)
Software remodularization consists of automatically reorganizing software packages to improve the overall program structure.

**Cohesion:** number of intra-edges

**Coupling:** number of inter-edges
• Bavota et al: Software Re-Modularization based on Structural and Semantic Metrics, 2010
  – Proposed an automated mono-objective where semantic and structural metric are combined in one objective value.

  – Propose an extension of its work with a Mono and Multi-Objective using an Interactive GA where the developer give their feedback to proposed remodularization solution.
• Harman et al: Software Module Clustering as a Multi-Objective Search Problem, 2011.
  – Use genetic algorithm with three objectives: Coupling, Cohesion and Complexity.

• Abdeen et al: Towards automatically improving package structure while respecting original design decisions, 2013.
  – Proposed a re-modularization task as a multi-objective optimization problem to improve existing packages structure while minimizing the modification amount on the original design.
Limitations

• Focus only on improving structural measures (cohesion, coupling, etc.)

• Violate the domain semantics

• Do not consider the number of code-changes (deviation from initial design) and development/maintenance history.

• Limited to only 2 types of changes
  • Move class
  • Split package
• Software remodularization as a many-objective search problem
  - 4 Structural measures (number of packages, number of classes per package, cohesion and coupling)
  - Semantic coherence (cosine similarity and Call graphs)
  - Number of operations per solution
  - Consistency with the history of changes

• New operation types
  - Move method
  - Extract class
  - Merge packages
  - Move class
  - Extract package
Approach Overview

- System to restructure
- Structural metrics
- Semantic metrics
- History of recorded changes

Many-objective Software Remodularization

Sequence of remodularization operations
The algorithm becomes unable to distinguish between solutions

Random search behavior

Require an additional selection process for convergence.
Multi-Objective issues

(K. Deb et al., ’13)

Generate initial Population N → Apply Mutation and Crossover → Generate Offsprings Q → Non-Dominated Sorting N+Q → Selecting M → If M = N → Niching → NO

Check stooping criteria → Yes → Solution

NO → Check stooping criteria → Yes → Solution
Each solution is represented as a vector of multiple refactorings

Each refactoring is generated randomly.

<table>
<thead>
<tr>
<th></th>
<th>Create Package(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Merge Package(p1,p2)</td>
</tr>
<tr>
<td>3</td>
<td>Split Package(p3, p4)</td>
</tr>
<tr>
<td>4</td>
<td>Move Class(C1, P1)</td>
</tr>
<tr>
<td>5</td>
<td>Merge Package(p5,p6)</td>
</tr>
</tbody>
</table>

Example of a remodularization solution

| Move Class(AttrNSImpl, org.apache.xerces.dom, org.apache.xerces.validators.dtd) |
| Extract Class(XGrammarWriter, XGrammarInput, parseInt())                          |
| Move Method(normalize(), XGrammarWriter, DTDGrammar)                               |
| Extract Package(org.apache.xerces.dom, org.apache.xerces.dtdl, CharacterDataImpl, ChildNode) |
• **Structural Metrics:**
  1. *number of classes per package*
  2. *number of packages in the system*
  3. *Coupling*
  4. *Cohesion*
• Semantic Metrics:

5. *Vocabulary-based similarity*:

\[
Sim(c_1, c_2) = \cos(c_1, c_2) = \frac{c_1 \cdot c_2}{\|c_1\| \ast \|c_2\|} \in [0,1]
\]

6. *Dependency-based similarity*:

\[
\text{sharedCallOut}(c_1, c_2) = \frac{|\text{callOut}(c_1) \cap \text{callOut}(c_2)|}{|\text{callOut}(c_1) \cup \text{callOut}(c_2)|} \in [0,1]
\]

\[
\text{sharedCallIn}(c_1, c_2) = \frac{|\text{callIn}(c_1) \cap \text{callIn}(c_2)|}{|\text{callIn}(c_1) \cup \text{callIn}(c_2)|} \in [0,1]
\]
Fitness Functions

• Other Metrics:

  7. Number of code changes: Sum of Operations

  8. Similarity with history of code changes

\[
Sim_{\text{history}}(RO) = \sum_{j=1}^{n} w_j
\]
Experiments

- 4 medium and large scale open systems and 1 industrial system provided by Ford Motor Company.

- Each experiment is repeated 32 times.

<table>
<thead>
<tr>
<th>Systems</th>
<th>Release</th>
<th># classes</th>
<th>KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces-J</td>
<td>v2.7.0</td>
<td>991</td>
<td>240</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>v6.1</td>
<td>585</td>
<td>21</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>v1.0.9</td>
<td>521</td>
<td>170</td>
</tr>
<tr>
<td>GanttProject</td>
<td>v1.10.2</td>
<td>245</td>
<td>41</td>
</tr>
<tr>
<td>JDI-Ford</td>
<td>v5.8</td>
<td>638</td>
<td>247</td>
</tr>
</tbody>
</table>
• Manual Precision:

\[ MP = \frac{\text{#coherent operations}}{\text{#proposed operations}} \in [0,1] \]
• **Automatic Validation**

\[
RE_{\text{recall}} = \frac{|\text{suggested operations} \cap |\text{expected operations}|}{|\text{expected operations}|} \in [0,1]
\]

\[
PR_{\text{precision}} = \frac{|\text{suggested operations} \cap |\text{expected operations}|}{|\text{suggested operations}|} \in [0,1]
\]

Results
How can our approach be useful for software engineers in real-world setting?
Challenges and Open Research Directions

• Why do we currently need to design special algorithms for each software engineering problem instance?
  – This is unrealistic: Science is about generality. Several software engineering activities have a lot of common patterns and similarities

• Why do we currently address silos of software engineering activity?
  – This is unrealistic: engineering decision making needs to take account of requirements, designs, test cases and implementation details simultaneously.
Challenges and Open Research Directions

- **Automation level:** How best do we draw the dividing line between adaptive automation for small changes and human intervention to invoke more fundamental adaption and to provide oversight and decision making?

- **Surrogate metrics:** Any approach that seeks dynamic adaptivity must necessarily compute many fitness evaluations between adaptations surrogate fitness computation will need to be fast.

- **Dynamic Adaptativity**
Take a Problem and “SBSE” it!

Thank You

Questions?