Automatic Inference of Structural Changes for Matching Across Program Versions

Miryung Kim, David Notkin, Dan Grossman
Computer Science & Engineering
University of Washington

Foo.mA()
Foo.mB()
Foo.mC()
Boo.mA(bool)
Boo.mB(bool)

Foo.mA(float)
Foo.mB(float)
Foo.mC()
Bar.mA(bool)
Boo.mA(int)
Boo.mB(int)
Code Matching Problem

<table>
<thead>
<tr>
<th>P</th>
<th>P'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar.Bar()</td>
<td>Bar.Bar()</td>
</tr>
<tr>
<td>Bar.mC(int)</td>
<td>Bar.mC(int)</td>
</tr>
<tr>
<td>Foo.mA()</td>
<td>Foo.mA(float)</td>
</tr>
<tr>
<td>Foo.mB()</td>
<td>Foo.mB(float)</td>
</tr>
<tr>
<td>Foo.mC()</td>
<td>Foo.mC()</td>
</tr>
<tr>
<td>Boo.mA(bool)</td>
<td>Bar.mA(bool)</td>
</tr>
<tr>
<td>Boo.mB(bool)</td>
<td>Boo.mA(int)</td>
</tr>
<tr>
<td></td>
<td>Boo.mB(int)</td>
</tr>
</tbody>
</table>
Our Approach: Matching with Change Rules
Our Approach: Matching with Change Rules

All methods in the Boo class take int argument instead of bool.
Motivations for Matching Code

- A fundamental building block for mining software repositories
- Also a basis for classic software evolution research and tools
  - Software version merging
  - Regression testing
  - Profile propagation
Matching is Challenging.

- Matching is hard due to code addition & deletion, copy & paste, refactorings, etc.
- Delta between two versions can be very large.
- For many uses, matching results must be concise and comprehensible.
Outline

- **background**
- our rule-based matching approach
- inference algorithm
- evaluation
- potential applications of change rules
Matching Problem $\approx$
Change Identification Problem

The problem of identifying code matches

↓  
The problem of identifying changes
Existing Approaches

diff, Syntactic Diff (CDiff), Semantic Diff, JDiff, origin analysis, refactoring reconstruction tools, etc.

Individually compare code elements at particular granularities using similarity measures
Limitations of Existing Approaches
Limitations of Existing Approaches

<table>
<thead>
<tr>
<th>P</th>
<th>P'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar.Bar()</td>
<td>Bar.Bar()</td>
</tr>
<tr>
<td>Bar_mC(int)</td>
<td>Bar_mC(int)</td>
</tr>
<tr>
<td>Foo_mA()</td>
<td>Foo_mA(float)</td>
</tr>
<tr>
<td>Foo_mB()</td>
<td>Foo_mB(float)</td>
</tr>
<tr>
<td>Foo_mC()</td>
<td>Foo_mC()</td>
</tr>
<tr>
<td>Boo_mA(bool)</td>
<td>Bar_mA(bool)</td>
</tr>
<tr>
<td>Boo_mB(bool)</td>
<td>Boo_mA(int)</td>
</tr>
<tr>
<td></td>
<td>Boo_mB(int)</td>
</tr>
</tbody>
</table>
Limitations of Existing Approaches
Limitations of Existing Approaches
Limitations of Existing Approaches

Cannot disambiguate among many potential matches
Limitations of Existing Approaches

Difficult to spot inconsistent and incomplete changes
Limitations of Existing Approaches

Output is an unstructured, usually lengthy list of matches
Limitations of Existing Approaches

- move axis drawing classes from chart to chart.axis
- add boolean input arg to all chart creation APIs

Output is an unstructured, usually lengthy list of matches
Limitations of Existing Approaches

Output is an unstructured, usually lengthy list of matches
Outline

✓ background

• **our rule-based matching approach**
• inference algorithm
• evaluation
• potential applications of change rules
Our Rule-based Matching Approach

• Our change rule can concisely describe a set of related refactorings and API changes at or above the method header level.

• Our tool automatically infers a set of likely change rules between two versions of a program.
Represent a high-level change pattern using a change rule

\[ \text{Easy to understand change intent} \]
Our Contribution 2. Conciseness

Concisely represent large deltas using a small number of change rules
Find matches evidenced by a more general change pattern

⇒ Improving recall
Our Contribution 4. Explicit Exceptions

for all x in Foo.m*() 
except \{Foo.mC()\} 
argAppend(x, float)

Our rule **encodes exceptions explicitly**

⇒ **Easy to notice** inconsistent and incomplete changes
Change Rule

for all x: method in scope transformation(x)
We use a regular expression to denote a set of methods

- e.g. `chart.Factory.create*Chart(*)`
Transformations At or Above the Level of Method Header

- 9 types of transformations representing:
  - replace the name of package, class, and method
  - replace the return type
  - modify the input signature, etc.
Change Rule with Exceptions

for all $x$:method in (scope - exceptions) 
transformation($x$)
Example Change Rule

Chart creation APIs were changed to take an additional \texttt{int} parameter.
Example Change Rule

For all x in Factory.create*Chart(*)
argAppend(x, [int])
Example Change Rule

For all $x$ in Factory.create*Chart(*) except \( \{\text{Factory.createPieChart()}\} \)
argAppend($x$, [int])
14 matches and 1 exception
Outline

✓ background
✓ our rule-based matching approach
• inference algorithm
• evaluation
• potential applications of change rules
Inference Algorithm Overview

Input: two versions of a program

Output: a set of likely change rules

1. Generate seed matches
2. Generate candidate rules by generalizing seed matches
3. Evaluate and select candidate rules (greedy algorithm)
Step 1: Generate Seed Matches

- Seed matches provide **hints** about likely changes.
- We generate seeds based on textual similarity between two method headers.
- Seed matches need not be all correct matches.
Step 2: Generate Candidate Rules for each seed \([x, y]\)

- Compare \(x\) and \(y\) and reverse engineer a set of transformations, \(T\).
- Based on \(x\), guess a set of scopes, \(S\).
- Generate candidate rules for each pair in \(S \times \text{PowerSet}(T)\).

Given a seed match,
\[
\text{[Foo.getBar(int), Boo.getBar(bool)]}
\]

**Transformations** = {
  replaceArg(x, int, bool)
  replaceClass(x, Foo, Boo)}

**Scopes** = \{\*.*(*), Foo.*(*), ..., *.get*(*), *.Bar(*), ..., Foo.get*(int), ... \}

**Candidate Rules** = {
  for all \(x\) in *.*(*)
    replaceArg(x, int, bool),
  for all \(x\) in Foo.*(*)
    replaceClass(x, Foo, Boo), ...,
  for all \(x\) in *.*(*)
    replaceArg(x, int, bool) AND
    replaceClass(x, Foo, Boo)\}
Step 3: Evaluate and Select Rules

- Greedily select a small subset of candidate rules that explain a large number of matches.
- In each iteration
  - evaluate all candidate rules
  - select a valid rule with the most number of matches
  - exclude the matched methods from the set of remaining unmatched methods
- Repeat until no rule can find any additional matches.
Finding Exceptions

a rule is **valid** if \( \# \) exceptions < \( \epsilon \times |\text{scope}| \)

For all \( x \) in `Factory.create*Chart(*)` 
argAppend\((x, [\text{int}])\)
Finding Exceptions

a rule is **valid** if \( \# \) exceptions < \( \varepsilon \times |\text{scope}| \)

For all \( x \) in `Factory.create*Chart(*)` except `{Factory.createPieChart}`
\[
\text{argAppend}(x, \ [\text{int}])
\]
3 matches 1 exceptions
Optimizations

- We **create** and **evaluate** rules **on demand**.

1. Candidate rules have subsumption structure. e.g. 
   \[ *.*.*(*\text{Axis}) \subset *.*.*(*) \]

2. The nature of greedy algorithm

- Running time: a few seconds (usual check-ins), average 7 minutes (releases)
Outline

✓ background
✓ our rule-based matching approach
✓ inference algorithm
• evaluation
• potential applications of change rules
Quantitative Evaluation

- Precision
- Recall
- Conciseness = \(|\text{Matches}| / |\text{Rules}|\) (M/R Ratio)

- We created evaluation data sets by manually inspecting our results combined with the results from other tools.
## Rule-based Matching Results for Three Release Archives

<table>
<thead>
<tr>
<th></th>
<th>JFreeChart</th>
<th>jHotDraw</th>
<th>jEdit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(17 release pairs)</strong></td>
<td>(4 release pairs)</td>
<td>(4 release pairs)</td>
<td></td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>94%</td>
<td>99%</td>
<td>93%</td>
</tr>
<tr>
<td>Median</td>
<td>(78~100%)</td>
<td>(82~100%)</td>
<td>(87~95%)</td>
</tr>
<tr>
<td><strong>Recall</strong></td>
<td>93%</td>
<td>99%</td>
<td>98%</td>
</tr>
<tr>
<td>Median</td>
<td>(70~100%)</td>
<td>(92~100%)</td>
<td>(95~100%)</td>
</tr>
<tr>
<td><strong>M/R ratio</strong></td>
<td>3.50</td>
<td>2.54</td>
<td>1.73</td>
</tr>
<tr>
<td>Median</td>
<td>(1.20~135.23)</td>
<td>(1.00~244.26)</td>
<td>(1.23~2.39)</td>
</tr>
</tbody>
</table>
Rule-based Matching Results for Three Release Archives

Top 20% of the rules find over 55% of the matches.
Top 40% of the rules find over 70% of the matches.
Comparison with Three Existing Tools

- UMLDiff [Xing and Strouliia 05]
- Refactoring Reconstruction [Weiβgerber and Diehl 06]
- Automatic Renaming Identification [S. Kim, Pan, and Whitehead 05]
## Comparison: Recall & Precision

<table>
<thead>
<tr>
<th>Programs</th>
<th>Other’s Recall</th>
<th>Our Recall</th>
<th>Other’s Prec.</th>
<th>Our Prec.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[XS05]</strong> jfreechart 18 releases</td>
<td>92%</td>
<td><strong>98%</strong></td>
<td>99%</td>
<td><strong>97%</strong></td>
</tr>
<tr>
<td><strong>[WD06]</strong> jEdit 2715 check-ins</td>
<td>72%</td>
<td><strong>96%</strong></td>
<td>93%</td>
<td><strong>98%</strong></td>
</tr>
<tr>
<td><strong>[KPW05]</strong> Tomcat 5096 check-ins</td>
<td>82%</td>
<td><strong>89%</strong></td>
<td>89%</td>
<td><strong>93%</strong></td>
</tr>
<tr>
<td><strong>[KPW05]</strong> jEdit 1189 check-ins</td>
<td>70%</td>
<td><strong>96%</strong></td>
<td>98%</td>
<td><strong>96%</strong></td>
</tr>
<tr>
<td><strong>[KPW05]</strong> ArgoUML 4683 check-ins</td>
<td>82%</td>
<td><strong>95%</strong></td>
<td>98%</td>
<td><strong>94%</strong></td>
</tr>
</tbody>
</table>
## Comparison: Recall & Precision

<table>
<thead>
<tr>
<th>Programs</th>
<th>Other’s Recall</th>
<th>Our Recall</th>
<th>Other’s Prec.</th>
<th>Our Prec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[XS05] jfreechart, 18 releases</td>
<td>92%</td>
<td>98%</td>
<td>99%</td>
<td>97%</td>
</tr>
<tr>
<td>[WD06]</td>
<td>2</td>
<td>98%</td>
<td>98%</td>
<td>93%</td>
</tr>
<tr>
<td>[KPW05] ArgoUML, 4683 check-ins</td>
<td>82%</td>
<td>95%</td>
<td>98%</td>
<td>94%</td>
</tr>
</tbody>
</table>

6-26% higher recall with roughly the same precision
## Comparison: Conciseness

<table>
<thead>
<tr>
<th></th>
<th>programs</th>
<th>Other’s Results</th>
<th>Our Results</th>
<th>Our Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[XS05]</strong> jfreechart</td>
<td>jfreechart 18 releases</td>
<td>4004 refactorings</td>
<td>939 rules</td>
<td>77% decrease in size</td>
</tr>
<tr>
<td></td>
<td>jEdit 2715 check-ins</td>
<td>1218 refactorings</td>
<td>906 rules</td>
<td>26% decrease in size</td>
</tr>
<tr>
<td></td>
<td>Tomcat 5096 check-ins</td>
<td>2700 refactorings</td>
<td>1033 rules</td>
<td>62% decrease in size</td>
</tr>
<tr>
<td><strong>[WD06]</strong> jEdit</td>
<td>jEdit 1189 check-ins</td>
<td>1430 matches</td>
<td>1119 rules</td>
<td>22% decrease in size</td>
</tr>
<tr>
<td></td>
<td>ArgoUML 4683 check-ins</td>
<td>3819 matches</td>
<td>2127 rules</td>
<td>44% decrease in size</td>
</tr>
</tbody>
</table>
## Comparison: Conciseness

<table>
<thead>
<tr>
<th>Programs</th>
<th>Other’s Results</th>
<th>Our Results</th>
<th>Our Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>[XS05] jfreechart 18 releases</td>
<td>4004 refactorings</td>
<td>939 rules</td>
<td>77% decrease in size</td>
</tr>
<tr>
<td>[WD06] 2715 check-ins</td>
<td>1218 refactorings</td>
<td>906 rules</td>
<td>26% decrease in size</td>
</tr>
<tr>
<td>[KPW05] ArgoUML 4683 check-ins</td>
<td>3819 matches</td>
<td>2127 rules</td>
<td>44% decrease in size</td>
</tr>
</tbody>
</table>

**22-77% reduction in the size of matching results**
Outline

- background
- our rule-based matching approach
- inference algorithm
- evaluation

- potential applications of change rules
  - bug finding, documentation assistant, API catch up, API evolution analysis, etc.
Potential App: Bug Finding Tool

for all x in J*.addTitle(Title) except {JThermometer.addTitle(Title)}
procedureReplace(x, addTitle, addSubtitle)

Dynamic dispatching of JFreeChart.addSubtitle does not work properly.
Conclusions

• **Matching** is a basis for a variety of software engineering research & tools.

• Our approach is the first to *automatically infer* structural changes and *concisely represent* them as a set of change rules.

• Our tool find matches with **high precision** and recall.
Acknowledgment

David Notkin
University of Washington

Dan Grossman
University of Washington

Zhenchang Xing
Eleni Strouliia
University of Alberta

Peter Weißgerber
Stephan Diehl
University of Trier

Sunghun Kim,
Jim Whitehead
University of California,
Santa Cruz