Most Influential Paper from ICSM 2010
“Template-based Reconstruction of Complex Refactoring” by Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim
What was 2010 like?

Earthquake, Port-au-Prince, Haiti

Air Travel Disruption from Volcanic Eruption, Iceland

ICSE 2010, South Africa, Cape Town
2nd year Assistant Professor: Miryung Kim

Deadlines sketched on the white board

Boxes of papers on book shelves
A fresh graduate from Vanderbilt U in 2010

RA offer

Miryung Kim <miryung@ece.utexas.edu> to Kyle, Stephanie →
Dear Kyle,

I have talked with your references and I am very excited about doing research with you! I am pleased to see your motivation and determination to pursue a Ph.D degree in software engineering and I believe our interests are very aligned.
A sophomore in Math and ECE in 2010

Do you know how to program? Do you know Java?

Could you please read this book and let me know your thoughts?
1st Year graduate Student: Nikita Sudan

A fresh graduate from U Maryland in 2010

---

applying to UTexas, Dr. Vibha Sazawal's student

Project-TemplaterRefactoring

Nikita Sudan <nsudan@gmail.com>
to miryung

Dear Dr. Kim,

Hope you are doing well. I am a senior Computer Engineering and Economics (double) major studying at the University of Maryland, College Park. I have been doing research under Dr. Vibha Sazawal, Assistant Professor, Department of Computer Science, UMD on the topic of Modeling Software Evolution using Game Theory. We recently submitted a paper to the International Conference on Software Processes, 2009.
ICSM 2010 in Timisoara

Analytics for Software Development

Thomas Zimmermann
Microsoft Research

ICSM 2010, Timisoara

http://thomas-zimmermann.com
Twitter: @tomzimmermann
What ideas have motivated and inspired RefFinder?
Dagstuhl: Multiversion Program Analysis in 2005
“How do we automatically match corresponding code elements between two program versions?”
Analyses of Software Evolution
- Evolution of Code Clones

High-level changes are often systematic at a code level

Automatic Inference of High-Level Change Descriptions
- Rule-based Change Representations
- Rule Learning Algorithms
Miryung’s PhD: Discovering Systematic Changes as Rules

<table>
<thead>
<tr>
<th>Changed Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Name</td>
</tr>
<tr>
<td>DummyRegistry</td>
</tr>
<tr>
<td>AbsRegistry</td>
</tr>
<tr>
<td>JRMPRegistry</td>
</tr>
<tr>
<td>JeremieRegistry</td>
</tr>
<tr>
<td>JacORBCosNaming</td>
</tr>
<tr>
<td>IIOPCosNaming</td>
</tr>
<tr>
<td>CmiRegistry</td>
</tr>
<tr>
<td>NameService</td>
</tr>
<tr>
<td>NameServiceManager</td>
</tr>
<tr>
<td><strong>Total Change:</strong> 9 files, 723 lines</td>
</tr>
</tbody>
</table>

- public class CmiRegistry implements NameService {
  + public class CmiRegistry extends AbsRegistry implements NameService {
    - private int port = ...
    - private String host = null
    - public void setPort (int p) {
      if (TraceCarol. isDebug()) { ...}
    - }
    - public int getPort() {
      return port;
    - }

Each rule represents **systematic changes** by relating groups of change facts. These rules are automatically inferred using **inductive logic programming**.

∀m ∀t past_method(m, “setHost”, t) ∧ past_subtype(“Service”, t) ⇒ deleted_calls(m, “SQL.exec”) [except t=”NameSvc” m=”NameSvc.setHost”]
Type-Oriented Logic Meta Programming
Kris De Volder

CodeQuest: Querying Source Code with DataLog
Elnar Hajiyev¹, Mathieu Verbaere¹, Oege de Moor¹ and Kris de Volder²

¹ Programming Tools Group
University of Oxford
United Kingdom

² Software Practices Lab
University of British Columbia
Vancouver, Canada

Navigating and Querying Code Without Getting Lost
Doug Janzen and Kris De Volder
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver BC Canada V6T 1Z4

Maintaining software through intentional source-code views
Kim Mens
Département INGI
Univ. catholique de Louvain
Louvain-la-Neuve, Belgium
Kim.Mens@info.ucl.ac.be

Tom Mens *
Programming Technology Lab
Vrije Universiteit Brussel
Brussels, Belgium
Tom.Mens@vub.ac.be

Michel Wermelinger †
Departamento de Informática
Universidade Nova de Lisboa
2829-516 Caparica, Portugal
mw@di.fct.unl.pt
### Detecting Merging and Splitting using Origin Analysis

### UMLDiff: An Algorithm for Object-Oriented Design Differencing

Zhenhong Yang and Fangli Streulin

### Automated Detection of Refactorings in Evolving Components

Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson

*Department of Computer Science*
*University of Illinois at Urbana-Champaign*

### Change Distilling: Tree Differencing for Fine-Grained Source Code Code Change Extraction

### SpyWare: A Change-Aware Development Toolset

Romain Robbes

Michele Lanza
Inspiration for RefFinder (3): Need for Domain Knowledge

- Statistical Relational Structure Learning
- Inductive Logic Programming
- Genetic Programming
- Heuristic Search

Infer too many “uninteresting” change rules
⇒ must encode inductive bias explicitly
Excerpts from Original
ICSM 2010 Talk
Motivation: Refactoring-Aware Code Review

• Developers can benefit from refactoring information when they investigate complex non-local edits during peer code reviews.

• Problem: How can we automatically identify the locations and types of refactoring from two program versions?
Challenges: Complex Refactoring Reconstruction

- Must find pre-requisite refactorings to identify composite refactorings
- Require information about changes within method bodies
- Require the knowledge of changes to the control structure of a program
Approach: Logic Query-based Refactoring Reconstruction

- Step 1. Encode each refactoring type as a template logic rule
- Step 2. Extract change-facts from two input program versions
- Step 3. Refactoring identification via logic queries
  - Ref-Finder orders pre-requisite refactorings before composite refactorings
# Predicates

<table>
<thead>
<tr>
<th>LSdiff Predicates</th>
<th>Extended Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>package</td>
<td>type</td>
</tr>
<tr>
<td>method</td>
<td>field</td>
</tr>
<tr>
<td>return</td>
<td>fieldoftype</td>
</tr>
<tr>
<td>typeintype</td>
<td>accesses</td>
</tr>
<tr>
<td>calls</td>
<td>subtype</td>
</tr>
<tr>
<td>inheritedfield</td>
<td>getter</td>
</tr>
<tr>
<td>inheritedmethod</td>
<td>addedparameter</td>
</tr>
<tr>
<td></td>
<td>conditional</td>
</tr>
<tr>
<td></td>
<td>cast</td>
</tr>
<tr>
<td></td>
<td>throws</td>
</tr>
<tr>
<td></td>
<td>methodmodifiers</td>
</tr>
<tr>
<td></td>
<td>parameter</td>
</tr>
<tr>
<td></td>
<td>similarbody(σ)*</td>
</tr>
<tr>
<td></td>
<td>fieldmodifiers</td>
</tr>
<tr>
<td></td>
<td>setter</td>
</tr>
<tr>
<td></td>
<td>deletedparameter</td>
</tr>
</tbody>
</table>
Fact-Level Differences

Old Program before_*

```
type("Foo",..)
method("Foo.main","main","Foo")
conditional("date.before(SUMMER_START)...")
methodbody("Foo.main", ...)
```

= ----------------- difference

New Program after_*

```
type("Foo",..)
method("Foo.main","main","Foo")
method("Foo.notSummer(Date)", "notSummer", "Foo")
```

= Differences (ΔFB) added_* / deleted_*

```
added_method("Foo.summerCharge", ...) 
added_method("Foo.notSummer", ...) 
deleted_conditional("date.before(SUMMER_START).
..)
```
Rule Syntax

Example: **collapse hierarchy** refactoring—a superclass and its subclass are not very different. Merge them together.

A rule’s consequent refers to a target refactoring to be inferred.

\[
(deleted_{\text{subtype}}(t_1,t_2) \\
\land (pull_{\text{up}}_{\text{field}}(f,t_2,t_1) \lor pull_{\text{up}}_{\text{method}}(m,t_2,t_1))) \\
\lor (before_{\text{subtype}}(t_1,t_2) \land deleted_{\text{type}}(t_1,n,p) \\
\land (push_{\text{down}}_{\text{field}}(f,t_1,t_2) \lor push_{\text{down}}_{\text{method}}(m,t_1,t_2)) \\
\Rightarrow collapse_{\text{hierarchy}}(t_1,t_2)
\]
Rule Syntax

Example: **collapse hierarchy** refactoring—a superclass and its subclass are not very different. Merge them together.

A rule’s antecedent may refer to pre-requisite refactorings.

\[
\begin{align*}
&\text{(deleted_subtype}(t_1,t_2) \\
&\land (\text{pull_up_field}(f,t_2,t_1) \lor \text{pull_up_method}(m,t_2,t_1)) \\
&\lor (\text{before_subtype}(t_1,t_2) \land \text{deleted_type}(t_1,n,p) \\
&\land (\text{push_down_field}(f,t_1,t_2) \lor \text{push_down_method}(m,t_1,t_2)) \\
&\Rightarrow \text{collapse_hierarchy}(t_1,t_2)
\end{align*}
\]
Encoding Fowler’s Refactorings

• We encoded 63 types but excluded a few because
  • they are too ambiguous,
  • require accurate alias analysis, or
  • require clone detection at an arbitrary granularity.
Collapse Hierarchy Inference

To find a move field refactoring

\[
\text{deleted_field}(f_1, f, t_1) \\
\land \text{added_field}(f_2, f, t_2) \\
\land \text{deleted_access}(f_1, m_1) \\
\land \text{added_access}(f_2, m_1) \\
\Rightarrow \text{move_field}(f, t_1, t_2)
\]

Fact-base

\[
\begin{align*}
\text{before_subtype}("Chart","PieChart") \\
\text{deleted_subtype}("Chart","PieChart") \\
\text{deleted_field}("PieChart.color","color","PieChart") \\
\text{added_field}("Chart.color","color","Chart") \\
\text{deleted_access}("PieChart.color","Chart.draw") \\
\text{added_access}("Chart.color","Chart.draw")
\end{align*}
\]
To find a **move field** refactoring

```prolog
move_field(f, t1, t2)
```

**Fact-base**

```prolog
before_subtype("Chart","PieChart")
deleted_subtype("Chart","PieChart")
deleted_field("PieChart.color", "color", "PieChart")
added_field("Chart.color", "color", "Chart")
deleted_access("PieChart.color", "Chart.draw")
added_access("Chart.color", "Chart.draw")
```
Invoke a **move-field** query

\[ \exists f_1, \exists f, \exists t_1, \exists t_2, \exists f_2, \exists m_1, \]
\[ \text{deleted_field}(f_1, f, t_1) \]
\[ \land \text{added_field}(f_2, f, t_2) \]
\[ \land \text{deleted_access}(f_1, m_1) \]
\[ \land \text{added_access}(f_2, m_1) \]
Collapse Hierarchy Inference

Create a new move field fact

\[
f = \text{"color"}, \\
t1 = \text{"PieChart"}, \\
t2 = \text{"Chart"} \\
move\_field(\text{"color"}, \text{"PieChart"}, \text{"Chart"})
\]

Fact-base

before\_subtype(\text{"Chart"}, \text{"PieChart"})
deleted\_subtype(\text{"Chart"}, \text{"PieChart"})
deleted\_field(\text{"PieChart.color"}, \text{"color"}, \text{"PieChart"})
added\_field(\text{"Chart.color"}, \text{"color"}, \text{"Chart"})
deleted\_access(\text{"PieChart.color"}, \text{"Chart.draw"})
added\_access(\text{"Chart.color"}, \text{"Chart.draw"})
move\_field(\text{"color"}, \text{"PieChart"}, \text{"Chart"})
To find a **pull up field** refactoring

\[
\text{move_field}(f, t1, t2) \\
\land \text{before_subtype}(t2, t1) \\
\implies \text{pull_up_field}(f, t1, t2)
\]

**Fact-base**

```plaintext
before_subtype("Chart","PieChart")
deleted_subtype("Chart","PieChart")
deleted_field("PieChart.color", "color", "PieChart")
added_field("Chart.color", "color", "Chart")
deleted_access("PieChart.color", "Chart.draw")
added_access("Chart.color", "Chart.draw")
move_field("color", "PieChart", "Chart")
```
To find a **pull up field** refactoring:

\[
\text{move\_field}(f, t1, t2) \\
\land \text{before\_subtype}(t2, t1) \\
\implies \text{pull\_up\_field}(f, t1, t2)
\]

**Fact-base**

- before\_subtype(“Chart”, “PieChart”)
- deleted\_subtype(“Chart”, “PieChart”)
- deleted\_field(“PieChart.color”, “color”, “PieChart”)
- added\_field(“Chart.color”, “color”, “Chart”)
- deleted\_access(“PieChart.color”, “Chart.draw”)
- added\_access(“Chart.color”, “Chart.draw”)
- move\_field(“color”, “PieChart”, “Chart”)

**Collapse Hierarchy Inference**

- **Collapse**
- **Pull Up**
- **Move**
Collapse Hierarchy Inference

Invoke a pull up field query

\[ \exists f, \exists t1, \exists t2, \]
\[ \text{move_field}(f, t1, t2) \]
\[ \wedge \text{before_subtype}(t2, t1) \]

Fact-base

- \text{before subtype(“Chart”, “PieChart”)}
- \text{deleted subtype(“Chart”, “PieChart”)}
- \text{deleted field(“PieChart.color”, “color”, “PieChart”)}
- \text{added field(“Chart.color”, “color”, “Chart”)}
- \text{deleted access(“PieChart.color”, “Chart.draw”)}
- \text{added access(“Chart.color”, “Chart.draw”)}
- \text{move field(“color”, “PieChart”, “Chart”)}
Collapse Hierarchy Inference

Create a new pull up field fact

```plaintext
Fact
f=“color”,
t1=“PieChart”,
t2=“Chart”
pull_up_field(“color”, “PieChart”, “Chart”)
```

Fact-base

```plaintext
before_subtype(“Chart”, “PieChart”)
deleted_subtype(“Chart”, “PieChart”)
deleted_field(“PieChart.color”, “color”, “PieChart”)
added_field(“Chart.color”, “color”, “Chart”)
deleted_access(“PieChart.color”, “Chart.draw”)
added_access(“Chart.color”, “Chart.draw”)
move_field(“color”, “PieChart”, “Chart”)
pull_up_field(“color”, “PieChart”, “Chart”)
```
Collapse Hierarchy Inference

Create a new collapse hierarchy fact

collapse_hierarchy("Chart", "PieChart")

Fact-base

before_subtype("Chart","PieChart")

**deleted subtype("Chart","PieChart")**
deleted_field("PieChart.color", "color", "PieChart")

added_field("Chart.color", "color", "Chart")
deleted_access("PieChart.color", "Chart.draw")

added_access("Chart.color", "Chart.draw")

move_field("color", "PieChart", "Chart")

**pull up field("color", "PieChart", "Chart")**
Collapse Hierarchy Inference

Create a new collapse hierarchy fact

Fact-base

before_subtype("Chart","PieChart")
deleted_subtype("Chart","PieChart")
deleted_field("PieChart.color", "color", "PieChart")
added_field("Chart.color", "color", "Chart")
deleted_access("PieChart.color", "Chart.draw")
added_access("Chart.color", "Chart.draw")
move_field("color", "PieChart", "Chart")
pull_up_field("color", "PieChart", "Chart")
collapse_hierarchy("Chart", "PieChart")
## Evaluation: Fowler’s

Ref-Finder finds refactorings with 97% precision and 94% recall.

<table>
<thead>
<tr>
<th>Types</th>
<th>Expected</th>
<th>Found</th>
<th>Precision</th>
<th>Recall</th>
<th>False negatives</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>8</td>
<td>19</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11-20</td>
<td>9</td>
<td>20</td>
<td>0.95</td>
<td>1</td>
<td></td>
<td>extract method</td>
</tr>
<tr>
<td>21-30</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31-40</td>
<td>10</td>
<td>13</td>
<td>1</td>
<td>0.9</td>
<td>preserve whole objects</td>
<td></td>
</tr>
<tr>
<td>41-50</td>
<td>9</td>
<td>11</td>
<td>1</td>
<td>0.89</td>
<td>replace conditionals with polymorphism</td>
<td></td>
</tr>
<tr>
<td>51-60</td>
<td>10</td>
<td>11</td>
<td>1</td>
<td>0.9</td>
<td>replace parameters with explicit methods</td>
<td></td>
</tr>
<tr>
<td>61-72</td>
<td>8</td>
<td>14</td>
<td>0.86</td>
<td>0.88</td>
<td>replace type code with state</td>
<td>replace magic number with symbolic constants, extract method</td>
</tr>
<tr>
<td>Total</td>
<td>63</td>
<td>100</td>
<td>0.97</td>
<td>0.94</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Ref-Finder finds refactorings with 74% precision and 96% recall.

<table>
<thead>
<tr>
<th>Versions</th>
<th># Found</th>
<th>Prec.</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>jEdit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.0-3.0.1</td>
<td>10</td>
<td>0.75</td>
<td>0.78</td>
</tr>
<tr>
<td>3.0.1-3.0.2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3.0.2-3.1</td>
<td>214</td>
<td>0.45</td>
<td>1</td>
</tr>
<tr>
<td>Columba</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300-352</td>
<td>43</td>
<td>0.52</td>
<td>0.9</td>
</tr>
<tr>
<td>352-449</td>
<td>209</td>
<td>0.91</td>
<td>1</td>
</tr>
<tr>
<td>Carol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62-63</td>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>389-421</td>
<td>8</td>
<td>0.63</td>
<td>1</td>
</tr>
<tr>
<td>421-422</td>
<td>147</td>
<td>0.64</td>
<td>0.9</td>
</tr>
<tr>
<td>429-430</td>
<td>48</td>
<td>0.85</td>
<td>1</td>
</tr>
<tr>
<td>430-480</td>
<td>37</td>
<td>0.81</td>
<td>1</td>
</tr>
<tr>
<td>480-481</td>
<td>11</td>
<td>0.91</td>
<td>0.9</td>
</tr>
<tr>
<td>548-576</td>
<td>20</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>576-764</td>
<td>14</td>
<td>0.85</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>774</td>
<td>0.74</td>
<td>0.96</td>
</tr>
</tbody>
</table>
Reflections on the paper
Example: Tools for Realistic Refactoring

- Accurate Refactoring Reconstruction
- Automated Change Documentation
- Multi-Objective Search-based Refactoring
- Refactoring-Aware Code Review and Merging
- Refactoring Error Detection
- Refactoring-Aware Testing and Dynamic Analysis
- Refactoring Recommendations
- Automated Clone Removal and Code Extraction
- Studies on Technical Debt, Code Smells, Refactoring Benefits

Refactoring Reconstruction
How We Refactor, and How We Know It

Emerson Murphy-Hill
Portland State University
emerson@cs.pdx.edu

Chris Parnin
Georgia Institute of Technology
chris.parnin@gatech.edu

Andrew P. Black
Portland State University
black@cs.pdx.edu

A Field Study of Refactoring Challenges and Benefits

Miryung Kim *
miryung@ece.utexas.edu

Thomas Zimmermann +
tzimmer@microsoft.com

Nachiappan Nagappan +
nachin@microsoft.com

Use, Disuse, and Misuse of Automated Refactorings

Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, Ralph E. Johnson
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{mvakili2, nchen, snegara2, rajkuma1, bpbailley, rjohnson}@illinois.edu

A Comparative Study of Manual and Automated Refactorings

Stas Negara, Nicholas Chen, Mohsen Vakilian,
Ralph E. Johnson, and Danny Dig
**WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings**

Stephen R. Foster  
*UC San Diego*  
La Jolla, CA  
srfoster@cs.ucsd.edu

William G. Griswold  
*UC San Diego*  
La Jolla, CA  
wgg@cs.ucsd.edu

Sorin Lerner  
*UC San Diego*  
La Jolla, CA  
imer@cs.ucsd.edu

**Reconciling Manual and Automatic Refactoring**

Xi Ge  
Quinton L. DuBose  
Emerson Murphy-Hill  
*Department of Computer Science, North Carolina State University, Raleigh, NC*  
{xge, qldubose}@ncsu.edu, emerson@csc.ncsu.edu

**RefDistiller: A Refactoring Aware Code Review Tool for Inspecting Manual Refactoring Edits**

Everton L. G. Alves†  †Myoungkyu Song†  †Miryung Kim†  
*University of Texas at Austin, USA†  †University of California, Los Angeles, USA§  †Federal University of Campina Grande, Brazil†  
{everton, mksong1117}@utexas.edu, miryung@cs.ucla.edu

**Manual Refactoring Changes with Automated Refactoring Validation**

Xi Ge  
Quinton L. DuBose  
Emerson Murphy-Hill
SE community took this work to several directions

- Accurate Refactoring Reconstruction
- Automated Change Documentation
- Multi-Objective Search-based Refactoring
- Refactoring-Aware Code Review
- Refactoring Error Detection
- Tools for Realistic Refactoring
- Refactoring-Aware Testing and Dynamic Analysis
- Studies on Technical Debt, Code Smells, Refactoring Benefits
- Refactoring Recommendations
- Automated Clone Removal and Code Extraction
RefFinder Tool Release

[ICSM’10, Prete et al. FSE’10 Demo, Kim et al.]
We propose to design, build, and evaluate a logic-query based refactoring reconstruction tool in Visual Studio.

1 Problem Statement
As software engineers collaboratively develop software, they often need to investigate code changes implemented by other developers. In order to improve developer productivity during peer code reviews, this research proposal focuses on a specific problem of refactoring reconstruction—How can we automatically identify the location and types of refactorings from two program versions?

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves the modular structure of software. While improving software maintainability, refactoring often involves coordinated edits to different parts of a system. For example, a replace conditional with polymorphism refactoring simplifies repetitive if statements by creating subclasses and by moving each leg of the conditional to an overriding method.

We hypothesize that developers can tremendously benefit from refactoring information inferred from code changes when they investigate non-local, complex edits. Recording refactorings in an IDE is inadequate alternative to this problem because developers often manually apply refactorings and behavior-modifying edits together. Murphy-Hill et al. find that almost 90% of refactorings are performed manually without the help of automated refactoring tools, and programmers do frequently intersperse refactoring with other program changes. In fact, pure refactorings only account for 15% of commits, while 28% of commits are refactorings combined with behavior-modifying edits [13].

Another reason why it is important to reconstruct refactorings from program versions is that manual implementation of refactoring is often error-prone. Weißgerber and Diehl investigated the rate of refactorings and bug fixes in open source projects and found that there is an increase in the number of bugs after refactorings [16]. In our recent study of three large open source projects (Eclipse, jEdit, and Columba), we found that the number of bug fixes increase by 12.4% in the 5 subsequent revisions after refactorings, and that these bug fixes often repair refactoring mistakes.

2 Background
Even though high-level program transformations such as refactorings are often systematic at a code level, existing program differencing tools such as diff produce low-level differences without much abstraction. Our prior work on logical program differencing [9, 8, 11] overcomes this limitation by raising the abstraction level beyond syntactic and textual differences using a logic rule-based approach.

Existing refactoring reconstruction techniques [2, 17, 18, 19] find only simple rename and move refactorings and cannot identify composite refactorings, which consist of atomic refactorings. Our approach, REFINDER, can infer complex refactorings by encoding each...
We integrate RefFinder with FaultTracer dynamic change impact analysis [ICSM’12]

While refactoring edits are only 8% of changes, 38% of affected tests are relevant to refactoring and a half of failed affected tests include refactoring edits.
Thankful to My Students

From Right to Left

Baishakhi Ray (PhD 2013 ⇒ Assistant Prof @ Columbia) Detecting Recurring Changes and Errors
Na Meng (PhD 2014 ⇒ Assistant Prof @ Virginia Tech) Automating Recurring Changes & Clone Removal
Tianyi Zhang (PhD 2019, Postdoc @ Harvard) Leveraging Redundancy for Code Review, Testing, API Usage Mining
Muhammad Ali Gulzar (PhD 2020 ⇒ Assistant Prof @ Virginia Tech) Debugging and Testing for Big Data Analytics
Myoungkyu Song (Postdoc 2015 ⇒ Assistant Prof @ Nebraska, Omaha) Error Detection in Refactoring Edits
Thankful to ICSME “Community”

ICSM 2009 Edmonton
My first PC

ICSM 2011 Williamsburg
My first OC/ERA co-chair

ICSM 2012 Riva del Garda

ICSM 2013 Einhoven

ICSME 2018 Madrid

ICSME 2019 my first
PC co-chair / my first SC
Most Influential Paper from ICSM 2010

“Template-based Reconstruction of Complex Refactoring”
by Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim