Ref-Finder: A Refactoring Reconstruction Tool based on Logic Query Templates

Miryung Kim, Matthew Gee, Alex Loh, Napol Rachatasumrit
The University of Texas at Austin
{miryung@ece, matt.gee@mail, alexloh@cs, nrachatasumrit@mail}.utexas.edu

ABSTRACT

Knowing which parts of a system underwent which types of refactoring between two program versions can help programmers better understand code changes. Though there are a number of techniques that automatically find refactorings from two input program versions, these techniques are inadequate in terms of coverage by handling only a subset of refactoring types—mostly simple rename and move refactorings at the level of classes, methods, and fields. This paper presents a Ref-Finder Eclipse plug-in that automatically identifies both atomic and composite refactorings using a template-based refactoring reconstruction approach—it expresses each refactoring type in terms of template logic queries and uses a logic programming engine to infer concrete refactoring instances. Ref-Finder currently supports sixty three types in the Fowler's catalog, showing the most comprehensive coverage among existing techniques.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*

General Terms

Algorithms, Design and Experimentation

Keywords

software evolution, refactoring, program differencing, logic-based program representation

1. INTRODUCTION

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. Automatically identifying which refactorings happened between two program versions is an important research problem because inferred refactorings can be used to study the role of refactorings during software evolution, to update client applications that are broken due to refactorings in library components [1], and to create more intelligent refactoring-aware merging and version control systems [2]. Our survey of existing refactoring identification techniques [7] found that, even though they can handle simple refactorings, such as *rename* and *move* refactorings, they cannot easily handle complex refactorings that consist of atomic refactorings related by structural constraints such as an *extract superclass* refactoring (pages. 336-340 [3]), which consists of *move fields/methods* refactorings.

To overcome these limitations, we devised a refactoring reconstruction approach that actively leverages the domain knowledge about well-known refactoring types [7]. As a first step, we targeted refactoring types in Fowler's catalog [3], a comprehensive list of refactorings that are well understood by software engineering practitioners. Inspired by the prior work on logic-based program representation approaches [6], we described the structural constraints before and after applying a refactoring to a program in terms of *template logic queries* and encoded ordering dependencies among refactoring types to define which refactorings types must be identified before identifying higher-level, composite refactorings.

This paper presents Ref-Finder which instantiates this logic query based refactoring reconstruction approach. Ref-Finder is an Eclipse plug-in that builds on top of our logical program differencing framework [4, 5]. It takes two program versions as input either from workspace snapshots or from a Subversion repository, and extracts logic facts about a program's syntactic structure using Eclipse Java Development Toolkit's AST analysis. Using the Tyruba logic programming engine [8], it then invokes pre-defined logic queries to identify program differences that match the constraints of each refactoring type under focus. The main contribution of this paper is a tool that *visualizes identified refactoring instances* within an existing Eclipse integrated development environment. The evaluation of Ref-Finder on code examples from Fowler's book, release pairs of jEdit, and revision pairs of Columba and Carol shows that its overall precision and recall are 79% and 95% respectively [7].

2. REF-FINDER FEATURES

A developer may begin her investigation by selecting two program versions either from her current workspace projects or revisions from a Subversion repository. Ref-Finder compares the syntax tree of each version to compute structural change-facts such as deleted_trycatch. Ref-Finder then invokes template logic queries, each of which encode the structural constraints of a program before and after refactorings.

Consider a *replace conditionals with polymorphism* refactoring example from version 4.3.1 of jEdit, an open source text editor. In the old version, the class `LHS` contained a method, `assign()`, whose behavior depended on the value of the `type` field. The new version of this methods supplants

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
ACM 978-1-60558-791-2/10/11.

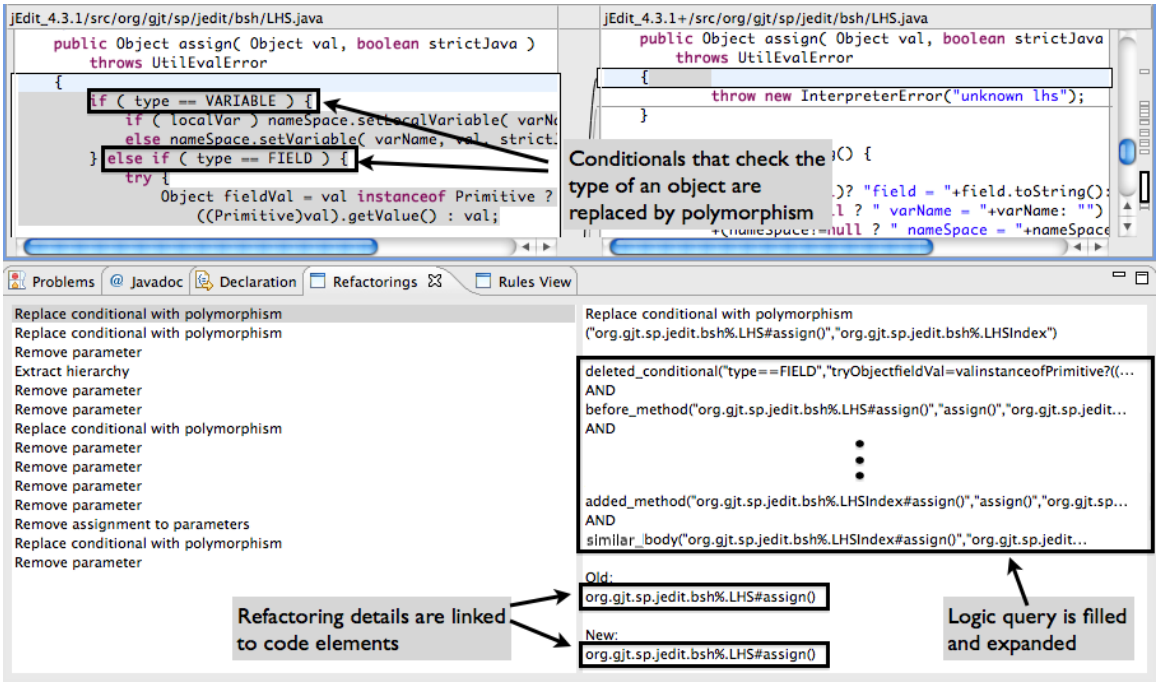

Figure 1: Ref-Finder **infers a** *replace conditionals with polymorphism* **refactoring from change facts** *deleted_conditional*, *after_subtype*, *before_method*, *added_method* **and** *similar_body*.

this conditional logic by adding LHS's subclasses and then using polymorphism to invoke a different behavior by overriding `assign()` in each of the subclasses. In order to find a replace_conditionals_with_polymorphism(oldmethod, subclass) refactoring, Ref-Finder invokes a query (deleted_conditional-(?condition, ?thenpart, ?elsepart, ?superclass) ∧ before_method-(?oldmethod, ?superclass), after_subtype(?superclass, ?subclass) ∧ added_method(?newmethod, ?subclass) ∧ similar_body(?new-method, ?oldmethod)), to check that a type check was performed in the conditional and that the new method body is similar to the original method. In our logic query description, ?x indicates an existentially quantified logic variable, x. As shown in Figure 1, Ref-Finder visualizes the reconstructed refactorings as a list. The panel on the right summarizes key details of the selected refactoring and allows the developer quickly navigate to the associated code fragments.

3. RELATED WORK AND SUMMARY

Existing refactoring reconstruction techniques compare code elements in terms of their name and structure similarity to identify move and rename refactorings [10, 1, 9]. The approach most similar to ours is Xing et al.'s *change-facts queries* [9]. Queries corresponding to well-known refactoring types are applied to the change-facts database to find concrete refactoring instances. According to our survey of 12 existing techniques [7], 40 out of 72 refactoring types in Fowler's catalog [3] are not covered by any of existing techniques. Ref-Finder currently supports 63 types, providing the most comprehensive coverage among existing tools.

Acknowledgment. The authors thank Kyle Prete for his contribution to the implementation and evaluation of Ref-Finder and Nikita Sudan's assistance in prototyping Ref-Finder. This research is in part supported by the National Science Foundation, under grant CCF-1043810.

4. REFERENCES

[1] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, pages 404–428, 2006.

[2] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07*, pages 427–436, 2007.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[4] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE '09*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[5] A. Loh and M. Kim. A program differencing tool to identify systematic structural differences. In *ICSE '10 Research Demo*, page 4, 2010.

[6] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. In *SEKE '02*, pages 289–296. ACM, 2002.

[7] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings (to appear). In *The 26th IEEE International Conference on Software Maintenance*, September 2010.

[8] K. D. Volder. *Type Oriented Logic Meta Programming*. PhD thesis, The University of British Columbia, 1998.

[9] Z. Xing and E. Stroulia. Refactoring detection based on umldiff change-facts queries. In *WCRE '06*, pages 263–274, Washington, DC, USA, 2006. IEEE.

[10] L. Zou and M. Godfrey. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.