Search-based Refactoring Detection

Rim Mahouachi, Marouane Kessentini
Computer Science Department
Missouri University of Science and Technology
Rolla, USA
{rimmah, marouanek}@mst.edu

Mel Ó Cinnéide
School of Computer Science and Informatics,
University College Dublin, Ireland
mel.ocinneide@ucd.ie

ABSTRACT
We propose an approach to automate the detection of source code refactoring using structural information. Our approach takes as input a list of possible refactorings, a set of structural metrics and the initial and revised versions of the source code. It generates as output a sequence of detected changes in terms of refactorings. In this case, a solution is defined as the sequence of refactoring operations that minimizes the metrics variation between the revised version of the software and the version yielded by the application of the refactoring sequence to the initial version of the software. We use and adapt global and local heuristic search algorithms to explore the space of possible solutions.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

Keywords
Search-based software engineering, software quality, refactoring, software metrics, heuristic search.

1. INTRODUCTION
Software systems are frequently refined and restructured for many reasons such as bug-fixing or source code modification to accommodate requirement changes. To perform these activities, one of the most widely used techniques is refactoring which improves design structure while preserving external behavior [14]. Many techniques to support refactoring have been proposed in the literature [14][15]**Error! Reference source not found.**. The majority of these techniques enable the application of manual or automated refactoring to fix design problems, e.g., bad smells.

A related but distinct problem arises when a software developer is faced with a version of an application that has been recently refactored. They may wish to comprehend what changes have occurred since the previous version, or the changes may require that other parts of the software be changed as well [4]. It would be very useful for them to know what refactorings have been applied to the previous version of the software to create the current, revised version. This is the problem we address in this paper, by using a stochastic search through the space of possible refactorings, using the metrics profile of the revised software to guide the search.

A number of existing approaches propose to detect changes between two (or more) software versions by composing atomic changes to refactoring operations such as adding and/or deleting program elements.

We distinguish between two categories in this existing work: the first category [5][6][7][8] detects only atomic differences (elementary refactorings) while the second category [11][12][13] is able to detect complex differences (composite refactorings). Our approach can be classified in the second category. In general, existing approaches propose to detect differences between software versions using pre- and post-conditions specified for each refactoring. In this case, the specified conditions are related to the possible changes that could be detected by comparing the source and revised code. However, it could be easy to detect explicit refactoring operations using pre- and post-conditions and then performing a code matching. However, composite refactorings that represent a composition of atomic operations are difficult to detect. In addition, the list of possible changes combination between models can be very large. Thus, it is a tedious task to specify conditions for each refactoring and possible code-change.

To overcome the above-mentioned limitations, we propose to consider the detection of refactorings between software versions as an optimization problem using structural metrics. Our approach takes as input a complete set of refactoring types and a set of software metrics, and generates as output a list of detected changes in terms of refactorings. In this case, a solution is defined as the sequence of refactoring operations that minimizes the metrics variation between the revised version of the software and the version yielded by the application of the refactoring sequence to the initial version of the software. Due to the large number of possible refactoring combinations, a heuristic method is used instead of an enumerative one to explore the space of possible solutions. Thus, we use and adapt a genetic algorithm as a global heuristic search. Genetic algorithms are a powerful heuristic search optimization method inspired by the Darwinian theory of evolution.

2. REFACTORING DETECTION BY STUDYING METRICS VARIATION
This section shows how the above-mentioned issues can be addressed and describes the principles that underlie the proposed method for detecting refactorings from structural information. Therefore, we first present an overview of the search-based algorithm employed and subsequently provide the details of the approach and our adaptation of a genetic algorithm to detect refactorings.

The general structure of our approach is introduced in Fig. 1. The approach takes as input the initial and revised source code, a set quality metrics and a complete set of refactoring types. The approach generates a set of refactorings that represents the evolution from the initial source code to the revised one. An Eclipse plug-in is used to calculate metrics values from the revised code version and the new version obtained after applying the proposed solution (refactoring sequence). The process of detecting refactorings can be viewed as the mechanism that finds the best way to combine refactoring operations of the input set of refactoring types, in order to minimize the dissimilarity between the metrics value of the revised code and the code that results from applying the detected refactorings.

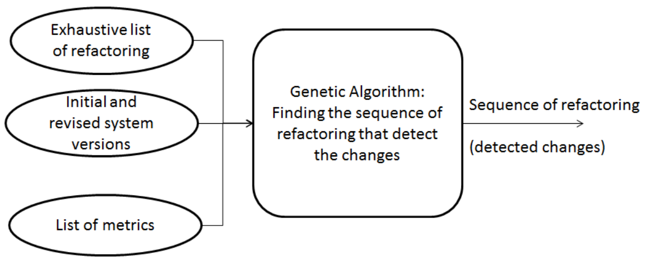
Exhaustive list of refactoring
Initial and revised system versions
List of metrics
Genetic Algorithm: Finding the sequence of refactoring that detect the changes
Sequence of refactoring
(detected changes)

Fig 1. Approach overview

Due to the large number of possible refactoring solutions, we consider the detection of refactoring between different software versions as an optimization problem. The algorithm explores a huge search space. In fact, the search space is determined not only by the number of possible refactoring combinations, but also by the order in which they are applied. To explore this huge search space, we use a global search by the use of a Genetic Algorithm (GA). This algorithm and its adaptation to the refactoring problem are described in the next section.

3. CONCLUSION

In this paper we introduce a novel, search-based approach to software refactoring detection between an initial software version and a refactored software version. Our approach is based on representing a proposed solution as a sequence of refactorings, and evaluating this solution in terms of its metrics profile as compared with the metrics profile of the refactored software version. Framing the problem in this manner enables us to use a Genetic Algorithm to evolve better solutions whose metric profiles more closely match that of the refactored software version. Our key hypothesis is that as the metric profiles converge, so too will the evolved refactoring sequence converge to the actual refactoring sequence that was originally applied to generate the refactored version from the initial version.

4. REFERENCES

[1] T. Ekman, U. Asklund, Refactoring-aware Versioning in Eclipse, Electronic Notes in Theoretical Computer Science 107 (2004) 57-69.

[2] R. Robbes, Mining a Change-Based Software Repository, in: Proceedings of the Workshop on Mining Software Repositories (MSR'07), IEEE Computer Society, 2007, pp. 15-23.

[3] M. Koegel, M. Herrmannsdoerfer, Y. Li, J. Helming, D. Joern, Comparing State- and Operation-based Change Tracking on Models, in: Proceedings of the IEEE International EDOC Conference, 2010.

[4] D. Dig, C. Comertoglu, D. Marinov, R. Johnson, Automated Detection of Refactorings in Evolving Components, in: ECOOP'06, Vol.4067 of LNCS, Springer, 2006, pp. 404-428.

[5] P. Weissgerber, S. Diehl, Identifying Refactorings from Source-Code Changes, in: Proceedings of ASE'06, IEEE, 2006, pp. 231-240.

[6] Identifying and Summarizing Systematic Code Changes via Rule Inference, Miryung Kim, David Notkin, Dan Grossman, Gary Wilson Jr. TSE: IEEE Transactions on Software Engineering.

[7] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, Miryung Kim: Template-based reconstruction of complex refactorings. ICSM 2010: 1-10

[8] S. Demeyer, S. Ducasse, O. Nierstrasz, Finding Refactorings via Change Metrics, in: Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'00), ACM, 2000, pp. 166-177.

[9] Z. Xing, E. Stroulia, Refactoring Detection based on UMLDiff Change-Facts Queries, in: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), IEEE, 2006, pp. 263-274.

[10] S. Vermolen, G. Wachsmuth, E. Visser, Reconstructing complex metamodel evolution, Tech. Rep. TUD-SERG-2011-026, Delft University of Technology (2011).

[11] J. M. Küster, C. Gerth, A. Förster, G. Engels, Detecting and Resolving ProcessModeling Differences in the Absence of a Change Log, in: Proceedings of the International Conference on Business Process Management (BPM'08), LNCS, Springer, 2008, pp. 244-260.

[12] Timo Kehrer, Udo Kelter, Gabriele Taentzer: A rule-based approach to the semantic lifting of model differences in the context of model versioning. ASE 2011: 163-172, (2011).

[13] M. Hartung, A. Gross, E. Rahm, Rule-based Generation of Diff Evolution Mappings between Ontology Versions, Computing Research Repository 1010.0122, (2010).

[14] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts: Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley, June 1999.

[15] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur: DECOR: A method for the specification and detection of code and design smells, Transactions on Software Engineering (TSE), 2009, 16 pages.