A Model-driven Approach to Formal Refactoring

Tiago Massoni
tlm@cin.ufpe.br

Rohit Gheyi
rg@cin.ufpe.br

Paulo Borba
phmb@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
Recife, Brazil

ABSTRACT

Applying refactorings to object-oriented systems usually affects source code and its associated models, involving complex maintenance efforts to keep those artifacts up to date. Most projects abandon design information in the form of models early in the life cycle, as their sustentation becomes extremely expensive. We propose a formal approach to consistently refactor systems in a model-driven manner. The refactoring applied to the model is linked to a sequence of behavior-preserving transformations that automatically refactor the underlying source code, based on structural properties from the model that must be implemented by the program. As a consequence, sound program refactoring can be accomplished without developer intervention, based only on the applied model transformations. Also, the refactored source code is consistent with the refactored model. Model information can be additionally used to improve refactoring automation, as more powerful transformations can be mechanized.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods; D.2.4 [**Software/Program Verification**]: Formal Methods

General Terms

Design, Verification

Keywords

Refactoring, Object Models, Program transformation

1. INTRODUCTION

Modern software evolution practices, such as *refactoring* [4], improve programs while maintaining their original behavior, in order to prepare software for change. Benefits of refactoring can be improved by the adoption of abstract models. However, little has been done for easing the task of developers when refactoring programs and consistently updating the resulting changes to the associated models. No support is offered to developers in order to safely refactor models and the underlying source code, ensuring semantic preservation. As a consequence, most projects abandon design information in the form of models early in the life cycle.

Figure 1 shows possible approaches for consistently evolving models and programs. Assuming that the program is refactored (Figure 1(a)), the associated structural model must be updated in order to maintain the desired consistency. Reverse engineering can generate a visualization of a program's structure, where M1, M2 denote models and P1, P2 denote programs. However, code visualization is cluttered by details inherent to implementation, usually restraining abstraction. In addition, it is usually unfeasible to extract structural design intent from code.

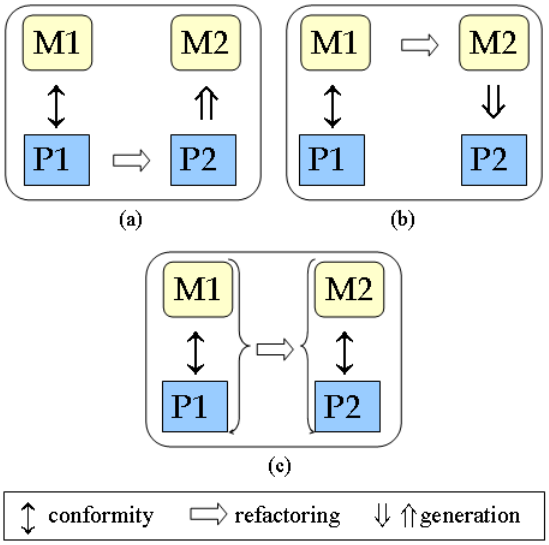

Figure 1: Approaches to consistent model and program refactoring

On the other hand, developers may refactor the model directly, as shown in Figure 1(b). After applying the refactoring, the underlying source code must be updated. Reflecting the structural changes to the program is usually supported by round-trip engineering. A usual technique marks previ-

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

ously edited code fragments as immutable, in order to avoid inconsistencies in non-generated code. Even so, this marked code might rely on refactored model structures, resulting in incorrect implementations after code regeneration. In this scenario, models and programs usually – and quickly – end up outdated.

2. GOAL STATEMENT

In this paper, we propose a formal approach to consistently refactor systems (showed in Figure 1(c)), by applying semantics-preserving transformations to *object models*. In particular, we consider object models in *Alloy* [3] and programs in a java-like formal language [1]. Alloy is a formal object modeling language, founded on relational logic that is similar to UML class diagrams annotated with OCL [5]. We apply the concept of primitive transformations, as they deal with one construct at a time, being possibly composed to form coarse-grained transformations, such as model refactorings.

Comprehensive sets of *primitive transformations* on object models [2] and object-oriented programs [1] form the basis of our approach. Each of these transformations is related to a sequence of semantics-preserving transformations on source code, based on structural properties guaranteed to be implemented by the program. Consequently, a formal relationship between refactoring at both levels is defined. In our approach, given a model is subject to refactoring, the conformity of the heap structure guarantees that the program can be safely refactored, with no further analysis, taking the program to the desired structure, an integrally model-driven process.

3. APPROACH AND EVALUATION

Our solution relies on an implementation notion between object models and programs. We consider that an object-oriented program implements a model when it presents a structural conformity and meets all of the specified model invariants throughout its execution. We delineate a notion of *structural conformity* in order to represent a mapping between models and programs, considering that the main structures (classes and relations) defined in the model must be declared by the program. In addition, model inheritance and collections are also enforced, yet in a flexible way. Given that the program meets this structural conformity and maintain the model invariants valid, these invariants are used to reason about the program's heap structure, being sufficient to determine analogous transformations in the program.

Invariants enable automatic program rewriting, as depicted by Figure 2. In order to prove the overall refactoring, we record this pattern of program transformations as a strategy, formalizing the sequence of program transformations as a single rule. We designate a program transformation rule for each model primitive transformation. Consequently, each transformation applied to a model indicates the sequence of transformations to be applied to the program. The invariants extracted from the model are involved in the program reasoning, as they must be valid under all circumstances, allowing for the *automatic application* of primitive transformations that rely on those properties.

Our approach expresses a way to mechanically reason about program refactoring. Such mechanization avoids manual updates, encouraging the maintenance of object models

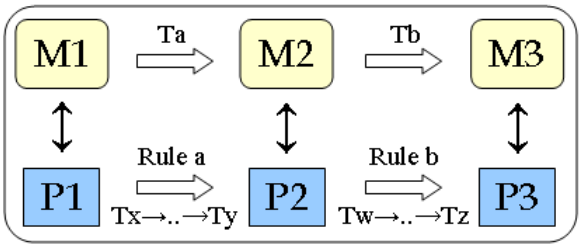

Figure 2: Our Approach to Model-driven Refactoring

when refactoring for evolution. Furthermore, this approach may advance tools that offer automatic refactoring for programs, since currently they do not employ any additional information beyond source code to perform transformations. In this case, the structural intent recorded in the model helped the definition of a more applicable refactoring. Our investigation also aims at advancing knowledge about the relationship between programs and its abstract representations as models, in order to improve evolutionary development based on abstract models. Limitations and assumptions of this study provide new evidences on the feasibility of model-driven methodologies, for example.

As a result of preliminary investigation, we formalized a number of rules for Alloy's primitive transformations. The results enlighten important issues relating object models and programs, such as translating model invariants regarding the program's heap, program rewriting based on those invariants and change of representation (*data refinement*). The rules will be formally proved in terms of *soundness* (semantic preservation) and *completeness* (applicable to any valid implementation of the model to which a sequence of primitive transformations was applied). We have applied a subset of the formalized rules in refactoring a small interpreter for Java expressions, previously modelled in Alloy. Future goals include validating the complete set of rules on real-world case studies.

4. REFERENCES

[1] P. Borba et al. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100, October 2004.

[2] R. Gheyi, T. Massoni, and P. Borba. Basic laws of object modeling. In *SAVCBS, at ACM SIGSOFT 2004/FSE-12*, pages 18–25, Newport Beach, United States, October 2004.

[3] D. Jackson et al. A Micromodularity Mechanism. In *Proceedings of the FSE/ESEC '01)*, pages 62–73. ACM Press, 2001.

[4] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[5] J. Warmer et al. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, second edition, 2003.