Toward a Formal Evaluation of Refactorings

John Paul, Nadya Kuzmina, Ruben Gamboa, and James Caldwell*

University of Wyoming,
Laramie, WY 82071-3315,
{jpaul, nadya, ruben, jlc}@cs.uwyo.edu

1 Introduction

Refactoring is a software development strategy that characteristically alters the syntactic structure of a program without changing its external behavior [2]. In this talk we present a methodology for extracting formal models from programs in order to evaluate how incremental refactorings affect the verifiability of their structural specifications. We envision that this same technique may be applicable to other types of properties such as those that concern the design and maintenance of safety-critical systems.

2 Formal Methodology

An object-oriented design D consists of a set of classes expressed in Java or another class-based object-oriented language. For the purposes of reasoning about D and formally comparing it to its refactorings we model D as a first-order theory of the form $\langle \Sigma, \mathcal{R} \rangle$ where Σ is a relational signature extracted from D's structural features, and $\mathcal{R}$ is a finite set of Σ-sentences expressing facts or axioms that partially capture D's class-level behavior [5]. $\mathcal{R}$ may result from the direct study of D or its documentation. We will consider the case when $\mathcal{R}$ is the output of a particular program analysis.

An additional set of Σ-sentences, $\mathcal{S}$, provides an abstract specification of what it means for D to be correct. The extent to which the set of facts $\mathcal{R}$, that we hold about D, implies $\mathcal{S}$, is indicative of how verifiable the design is by us. Any refactoring, D', of D will have the same correctness criteria as D. To compare the verifiability of the two designs we assume that there is a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ between the original and the refactored designs[1], but σ only needs to be defined on the subset of Σ used to express $\mathcal{S}$. Letting $\mathcal{R}$ and $\mathcal{R}'$ be the two sets of facts that we hold with respect to each design, we say that D' is better verifiable than D under the following conditions.

1. For every $\psi \in \mathcal{S}$, if $\mathcal{R}$ implies ψ, then $\mathcal{R}'$ implies the translated formula $\sigma(\psi)$.
2. For some $\psi \in \mathcal{S}$, $\mathcal{R}'$ implies $\sigma(\psi)$, but $\mathcal{R}$ does not imply ψ.

The first requirement merely states that D' is a behavior-preserving refactoring of D with respect to the verifiable behavior of D. While the second requirement states that we are able to verify D' more thoroughly than we are D.

* This material is based upon work supported by the National Science Foundation under Grant No. NSF CNS-0613919.

[1] More precisely the notion of a *derivor*[3] can be used.

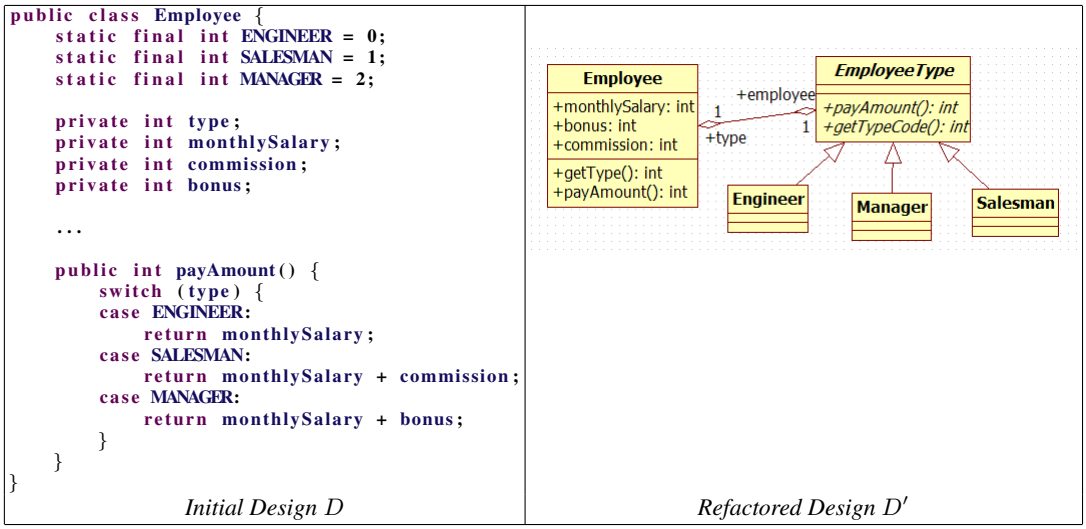

Fig. 1. Initial design D and refactored design D' of the *Employee* class.

Alloy [4] is a relational language based on first-order logic that allows us to express theories about designs. When combined with the Alloy Analyzer it offers a practical way to implement our methodology and to check 1) and 2) in practice. Once $\langle \Sigma, \mathcal{R} \rangle$ is presented as an Alloy theory, the Alloy Analyzer enumerates its models up to a user specified depth and reports any counterexample that it finds for a particular $\psi \in \mathcal{S}$, each of which is encoded as an Alloy assertion. The same is done for $\langle \Sigma', \mathcal{R}' \rangle$ and each $\sigma(\psi)$. While not a proof, the absence of a counterexample serves as evidence that an assertion may be valid within a theory and hence verifiable about a particular design.

3 Implementation

Next, we describe how we used Alloy and two different automatic constraint detectors to apply this methodology to the *Employee* example of the 'replace conditional with state' refactoring from Fowler's book [2]. Figure 1 presents the initial design D as a single class, `Employee`, providing a `payAmount` method which uses a `switch`-statement to compute the monthly earnings of an employee based on his or her occupation (`type`). In the refactored design D', however, `Employee` delegates the earnings computation to a polymorphic state object, `type`, which is no longer just a simple `int`, and the computation is now distributed over three different kinds of `EmployeeType` objects. The right side of Figure 1 depicts this design.

The extraction of Σ and Σ' is based on the class structures of D and D' and is virtually automatic. Each class or datatype is presented as its own disjoint set of atoms, while attributes and methods are presented as relations on these sets. The signature morphism $\sigma : \Sigma \to \Sigma'$ must be constructed by hand. In our example, σ is the identity mapping for all sorts and relations except for `type`. The assertions standing for the

specification $\mathcal{S}$ must also be constructed by hand. For instance, one assertion about the `payAmount` method is that an *Engineer*'s monthly earnings are equal to his or her salary.

Finally, $\mathcal{R}$ and $\mathcal{R}'$ consist of the machine translated invariants output by one of the two program analysis tools, Daikon [1] or ContExt. One fact that Daikon recovered from the design D' states that the `payAmount` method for the `Engineer` class returns the `monthlySalary` of its `Employee` attribute. After the two respective Alloy theories, $\langle \Sigma, \mathcal{R} \rangle$ and $\langle \Sigma', \mathcal{R}' \rangle$ have been created for designs D and D' by either Daikon or ContExt, we use the Alloy Analyzer to check whether the set of assertions that hold in the refactored theory is a proper superset of the assertions that hold in the unrefactored theory. If so, then we conclude that there is evidence to suggest that D' may be better verifiable than D in light of the facts obtainable by either tool.

In this example the Alloy analyzer considered all possible models consisting of two `Employee` entities, two `EmployeeType` entities and each `int` from -16 to 15. Our study suggests that the verifiability of the refactored design improves with respect to the facts obtained by Daikon, while the verifiability of either design is sufficiently good in light of the facts obtained by ConText.

4 Conclusions

Insofar as some structural properties of programs *are* safety-critical, the methodology presented here already applies to them. For instance, a specification for a controller may contain a safety-critical class invariant that states which configurations are reachable. Our methodology allows a way to monitor the verifiability of such properties as refactorings are applied throughout the software lifecycle. More investigation is needed to evaluate our approach on a real world example.

References

1. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.
2. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
3. J. A. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4. Prentice Hall, 1978.
4. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
5. V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In *Proceedings of Foundations of Software Egineering*, pages 207–216, Lisbon, Portugal, 2005.