Refactoring in Feature-Oriented Programming: Open Issues

Ilie Şavga Florian Heidenreich

Institut für Software- und Multimediatechologie, Technische Universität Dresden, Germany
{ilie.savga, florian.heidenreich}@tu-dresden.de

Abstract

Similar to refactoring, feature-oriented programming can be seen as a metaprogramming paradigm, in which programs are values and composition operators transform programs to programs. In this position paper we discuss open issues of applying refactoring in the context of feature-oriented programming. First, we elaborate on the role of refactoring in maintaining features and their implementations as well as the impact of refactoring on the relation between the *problem* and *solution spaces*. Second, we discuss issues of relating well-known refactoring formalisms to existing formal approaches used in feature-oriented programming. Third, we suggest to use refactoring semantics to upgrade and test final products of a product line.

1. Introduction

Research in the area of Software Product Lines (SPL) focuses on the design and automatic synthesis of product families (11). An important concept in this area is that of a *feature*–a refinement in the product functionality (33). Each product within a product family can be identified by a unique combination of features, from which it is created. By modeling the *problem space* of a domain, a *feature model* defines all legal feature configurations. A particular configuration chosen by the user is used to generate a final product out of feature modules that comprise the *solution space* (12).

Feature-oriented programming (FOP) is concerned with designing and implementing features and can be seen as a metaprogramming paradigm (6): feature composition modifies (by addition and extension) base programs using features. Another well-known metaprogramming paradigm is *refactoring*—program-to-program transformations that "does not alter the external behavior of the code yet improves its internal structure." (18, p.9) Several recent publications (e.g., (6), (9), (28), (40)) point out various contexts where refactorings and FOP overlap. Inspired by their observations, in this position paper we elaborate on open issues relating FOP and refactorings. In particular, we overview related work and discuss:

- Refactoring software artifacts in FOP (Section 2). In FOP refactoring means restructuring software artifacts of the solution space, the problem space, or both. When refactoring the solution space, how to keep multiple representations of a feature consistent? When refactoring the problem space, what is a meaningful library of refactoring operators to restructure a feature model? Moreover, in some cases refactoring one space requires refactoring the another space. How could one synchronize both spaces to preserve their consistency?
- Refactoring definition in a formal FOP calculus (Section 3). Existing work on refactoring formalization varies in the way refactorings are defined. Which formalism is more appropriate to be integrated into existing FOP formal definitions? And which FOP formalism, if any, would be appropriate for such integration?
- Refactoring-based product maintenance (Section 4). As refactorings are formally defined, the refactoring history of a component can be treated as a formal specification of its change representing a kind of *maintenance delta* (10) to be used for automatic software construction and maintenance. In the context of FOP, how to use formal specification to alleviate maintenance tasks, such as upgrade and testing, of final feature-based software products?

Our main goal is to foster discussions on the role of refactoring in developing and maintaining software artifacts in the context of FOP.

2. Refactoring Software Artifacts in FOP

Extensively used software artifacts have to evolve considerably, because, according to the Lehman's first law, "a large program that is used undergoes continuing change or becomes progressively less useful." (30, p. 250) The second

McGPLE'08 October 23, 2008, Nashville, Tennessee, USA.
ACM [to be supplied].

Lehman's law says, that "as a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it." (30, p. 253). This work is performed by program restructuring, called refactoring in object-oriented systems (32).

Since both the problem and the solution spaces are represented by software artifacts, evolving the latter will inevitably imply their refactoring.[1]

2.1 Refactoring Solution Space

An example of refactoring the solution space is refactoring the source code of a feature module to address changed security requirements. Developers may need to move some functionality previously accessible in more general features to more specialized features restricting the visibility of functionality to more specific features. This operation will involve moving members among feature modules and is an example of a reactive refactoring. An example of a proactive refactoring is renaming certain members of feature modules to follow established naming conventions improving thus future maintainability of modules.

In general, besides source modules, features can be implemented by modules containing, for instance, binary code, grammars and makefiles (8), XML documents (2) or UML diagrams (13). Therefore, in addition to code refactoring (e.g., (15; 18; 32; 34)), in the context of FOP one should also consider existing work on refactoring of other software components, such as sequence and protocol state machines (36) or UML class diagrams (37). A practical problem is how to correlate refactorings in multiple representations of the same feature.

A possible solution is to use functions that map refactorings of one representation to refactorings of other representations (similar to *mapdeltas* as proposed by Batory (7)). It is, however, not clear whether it is possible to find a refactoring counterpart in any possible representation, that is, to map refactorings in all representations used in the solution space. This problem stems from the lack of rigor in formal specifications of refactorings for certain representations (e.g., what is a grammar refactoring?). Moreover, as Batory mentions (7), creating and maintaining functions that relate transformations may burden programmers considerably.

2.2 Refactoring Problem Space

Facing new and changing requirements, besides evolving the solution space, the variability of a product line has to evolve, too. If not refactored, features of a feature model will not reflect changing domain requirements. Moreover, the number of features may explode making the model unmanageable.

In a large industrial case study, Loesch and Ploedereder (31) show how formal concept analysis (19) of existing products can be used to find obsolete (unused) features as well as to derive missing feature constraints. Their restructuring proposals, such as merge/remove variable features and mark mutually exclusive features as alternatives, can be considered feature model refactorings.

Alves et al. (1) define a feature model refactoring as a transformation that improves the model's configurability (i.e., the number of valid variants defined by the model). Based on their definition, the authors suggest a number of refactorings that either increase configurability of the model (while addressing new model requirements or merging feature models) or does not affect the number of model's variants (while maintaining the model).

As opposed to enlarging the number of model's variants, Kim and Czarnecki (29) discuss the impact changes made to a feature model may have on model's specializations (i.e., models derived successively by specializing the initial model). For several changes applied to the initial model (e.g., cardinality change or feature addition), the authors define synchronizing changes in specializations, including the final specialization (i.e., configuration). Changes mentioned by Kim and Czarnecki (29) can also be seen as refactorings operating on feature models.

A future research direction is reusing the work of (1; 31; 29) to define a set of feature model refactorings, such as `MakeMandatory`, `MakeAlternative`, `DeleteFeature`, `CopyFeature` or `ReduceGroupCardinality`. For that, an important requirement is to define precisely preconditions and, perhaps, synchronization actions of such refactorings for relating changes of the problem space to the solution space (as discussed in the next section).

2.3 Synchronizing Refactored Spaces

The key issue in refactoring of the two spaces is that their refactoring should not be considered in isolation; otherwise seemingly safe changes may lead to wrongly composed final products. For example, consider refactoring the solution space that makes one feature module dependent on another module.[2] If applied only to the solution space, this valid (with regard to the module implementation) refactoring is not reflected in the feature model as an additional constraint between the features involved. As a consequence, the model will permit a configuration that includes the depending feature without the feature it depends on. As another example, if a feature in the feature model is made optional, whereas other features depend on its functionality in their implementation, the feature may be missing in a configuration leading to invalid final product implementation. As an extreme case, for a wrongly defined feature model the set of its configurations may be empty.

In general, changes made to one space should propagate to another space. More precisely, in case refactorings change

[1] Terminology note: in this section by refactoring of problem and solution spaces we mean refactoring of feature models and their corresponding feature modules as opposed to *feature-based refactoring* of legacy software into a set of feature modules (e.g., (3; 27; 28; 40)).

[2] Our examples are inspired by Czarnecki and Pietroszek (13) and Thaker et al. (38)

constraints on the valid combination of features or feature modules, constraints of another space must be updated correspondingly. It is important that the overall "strictness' of the implementation constraints of the solution space and the domain constraints of the problem space are not equivalent. The implementation constraints are defined by the language (metamodel), in which feature modules are defined, and by the modules' implementation itself. Domain constraints are defined by the feature model. In general, as argued by Czarnecki and Pietroszek (13) and Thaker et al. (38), the domain constraints must imply the implementation constraints. Representing constraints in propositional formulas and using SAT solvers, it is possible to automatically detect when such implication does not hold, and then manually solve inconsistencies between the two spaces (13; 38).

The elegant solution of using SAT solvers is, however, not perfect in the context of agile refactoring of problem and/or solution spaces. Drawing analogies with conventional code refactoring, it would mean manually changing the program, recompiling it to detect possible problems and then solving the latter manually. Instead, when applying small changes to features or their implementation, it would be preferable to immediately know whether changes are safe and do not lead to space inconsistencies. Moreover, similar to a refactoring engine updating calls to a renamed method or prompting for a default value of a new parameter, some space inconsistencies could be interactively fixed, at least, semi-automatically.

Generalizing this discussion, an important issue with regard to refactoring in FOP is how to synchronize the spaces being refactored to ensure no invalid product may be generated afterwards. With this regard, two important issues to be considered are:

1. How the relation between the two spaces is defined. Space relation may be defined as feature template annotations (13), code annotations relating features to AST nodes (28)[3], a separate metamodel-based specification (20), or a systematically organized directory and file structure (8).
2. Which safeness constraints (in other words, invariants that these constraints preserve) a refactoring must respect. Safeness constraints may be defined by separate specifications (e.g., OCL expressions (13), propositional formulas (38) or type rules (28)) or may be embedded into the language type system (4; 27).

The space relation complemented with safeness constraints can be treated as a model to reason about and detect space inconsistencies. Janota and Botterweck (26) explicitly define such a model, which they call feature-component model, by formally specifying the feature model, component model and constraints on their relations. They derive the feature model induced by a feature-component model, compare it with the provided feature model and detect possible weaknesses of the latter.

Motivated by the aforementioned work, an important question is how to define and realize an interactive refactoring environment permitting for safe refactoring of the problem and solution spaces and semi-automatic space synchronization.

3. Refactoring Definition in a Formal FOP Calculus

Recent work on formalization of feature-based software development (5; 9) aim for an algebra to represent and reason about features and their composition.[4] The key idea is of both approaches is the same: find an atomic unit of feature representation, use it to uniformly define feature structure and then define precisely how those structures are composed. Moreover, feature composition is always feature addition and/or feature modification. Feature modification is performed by modifiers—`(selector,rewrite)` pairs applied to atomic units, where `selector` finds program units (using pattern matching) and then `rewrite` applies to the units found. However, the two aforementioned approaches differ in the way they model features and divide composition power between addition and modification.

Batory and Smith (9) use as the atomic unit of feature representation a (primitive) term, that is, a key-value pair. A features is either a vector of terms or a delta vector. The latter is a unary function that transforms vectors to vectors by addition and modification. Feature addition uses set union (using term names) of vector terms and raises an error when conflicting term names occur. Feature modification is term selection and term rewriting.

In the feature algebra of Apel et al. (5), the atomic representation unit is a tree node (a so-called *atomic introduction*). As a consequence, a feature is modeled as a tree, called a *feature structure tree* (FST) of various abstraction levels. Feature addition is tree superimposition (i.e., node conflicts are resolved by language-specific overriding of leaf nodes).[5] Feature modification is tree traversal and tree rewrite. The key difference to the work of Batory and Smith (9) is the shift of composition power from modification to addition: due to overriding, such concepts as mixins can be unified with introduction (performed by union set) and need not be modeled by modifiers, as in (9).

The vision of Apel et al. (5) and Batory and Smith (9) is that by implementing an uniform calculus one will develop

[3] On the contrary to feature composition, in CIDE (28) annotations are used for feature *decomposition* to support feature-oriented refactoring of existing programs into features.

[4] The algebra of Höfner et al. (23) focuses on the analysis phase of feature-oriented development and considers neither the structure of features nor their implementation. Since the latter two are our main concerns regarding refactoring, we do not discuss the aforementioned work in this paper.

[5] More exactly, FST can be seen as a set of superimposed (added) atomic introductions and superimposition is modeled by the operator *introduction sum* composing introductions, hence FSTs.

an object-oriented framework for feature composition that is independent of a concrete implementation language. In such framework, all kinds of manipulated programs are represented uniformly under a common superclass. This superclass provides standard operations (implemented specifically for each supported kind of programs) to query and transform programs. For us it is interesting to investigate how existing work on program refactoring can be transferred to such uniform frameworks of program transformations. With this regard, the dual research questions is 1) which existing refactoring formalism could be adopted easier into a FOP calculus, and 2) which FOP calculus, if any, is appropriate for such adoption.

While classical refactoring definitions (32; 34) use predicate logic to define preconditions, in the context of tree manipulation more recent work using graphs to define preconditions, either by testing of presence conditions (42) or by defining a graph pattern to be matched (22; 41), may be more appropriate for defining pattern matching used by modifiers. Moreover, while the actual transformation of a refactoring is usually described informally (18; 32; 34; 39; 42), for uniform transformations of program graphs one should also consider formal definition of refactoring transformations using graph rewriting (22; 41) to define `rewrite` of modifiers.

Batory and Smith define two types of modifiers (9). While a *universal modifier* finds all program terms that have the name or value `selector` (and rewrite these terms), an *existential modifier* attempts to find `selector` by name. If the term is undefined, it assigns the term an existence error; otherwise, the modifier does nothing (9). Whereas probably all refactorings require universal modifiers (for example, to rewrite multiple method calls or push down methods to several subclasses at once), the question is whether some refactorings may also require existential modifiers. For instance, it may make sense to test for the method usage before deleting it and signal an error in case it is in use. In a sense, it would be similar to the code analysis implemented in the conventional refactoring engines, but at the general and uniform level of the finite map space.

Finally, and most important, the key difference of a refactoring from a feature is that refactoring may also require term (node) deletion. For example, moving a method can be seen as deleting it from one class and adding to another class. Future work mentioned by Batory and Smith (9) is defining delta vectors that support vector subtraction. However, allowing such modification to parts of features may lead to a situation that the target of a subsequent feature composition is eliminated by a previously executed refactoring. An appealing research direction is to investigate and define a proper interaction (composition) of features and refactorings.

4. Refactoring-based Product Maintenance

Since refactorings are program transformations with formally defined semantics (32; 34), a history of refactorings applied to a component can be treated as a formal specification of the component's structural change (16). Such specification could be used to alleviate tasks of maintaining a product line, for example, upgrading and testing.

4.1 Module Upgrade

After a feature module is refactored, one may want to also update existing products to propagate improvements of the new module version. In some cases, simple recompilation of all modules would take too much time and their complete redeployment. Instead, it may be preferred to update only the refactored modules.

Several software engineering approaches (17; 21; 35) use refactoring history to automatically upgrade a software library (or a framework). They base on the fact that more than 80% of library changes that break library-dependent applications are API refactorings (16). Using refactoring information, it is possible to adapt existing applications to the new library version (21), the new library to existing applications (17) or create adapters that translate between the library and its applications (35).

In the line of these approaches, refactoring semantics can be used to upgrade existing products of a product line. For example, similarly to the approach of Henkel and Diwan (21) refactorings could be effectively re-executed on the product implementation, synchronizing it with the refactored implementation. As another example, using refactoring history it would be possible to partially decompose an existing product extracting obsolete feature implementation and then compose back the final product using refactored feature implementation.

4.2 Product Line Testing

To ensure that generated feature-based programs are correct, Batory (7) suggests to use specification-based product line testing using Alloy (25). An Alloy specification describes properties of the program to be verified. Out of this specification, a set of input tests represented by a propositional formula is generated, solved by a SAT solver and converted into a test (7).

Although we do not have any practical results, we envisage using refactoring semantics to automatically derive such specifications (and, hence, product line tests). Several formalisms of refactoring definition use the notion of postconditions for refactoring definitions, either as logic predicates (34) or as modified graphs (41; 42). Although actual definitions differ, the intuition is the same: a postcondition reflects the semantics of the refactoring transformation and describes important structural particularities of the refactored program. An approach would be to convert the postconditions into a propositional formula and generate tests in the

similar manner as Batory suggests (7). This would detect program errors introduced, for example, by bugs in refactoring engines (14). Furthermore, depending on how the invariants are reflected by postconditions, it could also detect "bad smells" specific for FOP, like inadvertently overriding a method in a base class by a renamed method in a feature refining that base class. Moreover, because refactorings can be composed (34), given a refactoring history as a sequence of refactorings its composed postcondition can be derived. In such cases, there is no need to create tests for each single refactoring—a set of tests can be created at once for the whole refactoring history.

To guarantee type-checking software produce lines for the price of reduced language power, Kästner and Apel (27) adopt the Featherweight Java (FJ)–a formally specified minimal functional subset of Java (24)–as a implementation language of product lines. They propose Color Featherweight Java (CFJ) as a FJ-based calculus to describe the entire (valid) software product line in combination with annotations and prove that, if the product line is well-typed (with regard to the CFJ language grammar), then all generated FJ variants will be well-typed (i.e., the generation will preserve typing). When using such a language subset as FJ with proved type-soundness, a question is which transformations considered refactorings for its original superset (i.e., Java) can be considered as such for FJ, that is, do not lead to typing errors (and invalid program variants) according to the grammar of FJ.

5. Summary

In our position paper we discuss open issues of applying refactoring in the context of FOP. We believe that by addressing the research and practical questions formulated in the paper, it is possible to integrate existing work on program refactoring into the context of FOP building a framework for uniform program transformation and tools that combine feature-oriented programming and refactoring. With regard to the inherent complexity of developing software product lines, it is important that these tools will foster agile development and maintenance of feature-related software artifacts and will give a uniform view on a feature-oriented development environment.

Acknowledgements

This research has been co-funded by the German Ministry of Education and Research (BMBF) within the project feasiPLe (cf. `http://www.feasiple.de`).

References

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 201–210, New York, NY, USA, 2006. ACM.

[2] F. I. Anfurrutia, O. Díaz, and S. Trujillo. On the Refinement of XML. In *ICWE'07: Proceedings of the Seventh International Conference on Web Engineering*, volume 4607 of *Lecture Notes in Computer Science*, pages 473–478. Springer, 2007.

[3] S. Apel, C. Kästner, and D. Batory. Program refactoring using functional aspects. In *GPCE'08: Proceedings of the Seventh International Conference on Generative Programming and Component Engineering*, Nashville, Tennessee, USA, October 2008. ACM.

[4] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement:. In *GPCE'08: Proceedings of the Seventh International Conference on Generative Programming and Component Engineering*, Nashville, Tennessee, USA, October 2008. To appear.

[5] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In J. Meseguer and G. Rosu, editors, *AMAST'08: 12th International Conference on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, pages 36–50. Springer, 2008.

[6] D. Batory. Program refactoring, program synthesis, and model-driven development. In *CC'07: Proceedings of the 16th International Conference on Compiler Construction*, volume 4420 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2007.

[7] D. Batory. From practice towards theory: Using elementary mathematics as a modeling language. In *GPCE'08: Proceedings of the Seventh International Conference on Generative Programming and Component Engineering*, Nashville, Tennessee, USA, October 2008. To appear.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. *IEEE Transactions in Software Engineering*, 30(6):355–370, 2004.

[9] D. Batory and D. Smith. Finite map spaces and quarks: Algebras of program structure. Technical Report TR-07-66, University of Texas at Austin, Dept. of Computer Sciences, 2007.

[10] I. D. Baxter. Design maintenance systems. *Commununication of the ACM*, 35(4):73–89, 1992.

[11] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[12] K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools and Applications*. Addison-Wesley, Boston, MA, 2000.

[13] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE'06: Proceedings of the Fifth International Conference on Generative Programming and Component Engineering*, pages 211–220, New York, NY, USA, 2006. ACM.

[14] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC-FSE'07: Proceedings of the the Sixth Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[15] D. Dig. *Safe Component Upgrade*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, October 2007.

[16] D. Dig and R. Johnson. The role of refactorings in API evolution. In *ICSM'05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[17] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 441–450, New York, NY, USA, 2008. ACM.

[18] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[19] B. Ganter and R. Wille. *Formal Concept Analysis - Mathematical Foundations*. Springer, 1996.

[20] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping features to models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 943–944, New York, NY, USA, May 2008. ACM.

[21] J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM Press.

[22] B. Hoffmann, D. Janssens, and N. van Eetvelde. Cloning and expanding graph transformation rules for refactoring. In *Electronic Notes in Theoretical Computer Science*, volume 152, pages 53–67, Mar. 2006.

[23] P. Höfner, R. Khedri, and B. Möller. Feature algebra. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2006.

[24] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[25] D. Jackson. Alloy: A leightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[26] M. Janota and G. Botterweck. Formal approach to integrating feature and architecture models. In *FASE'08: Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2008.

[27] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *ASE'08: the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, September 2008.

[28] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-independent safe decomposition of legacy applications into features. Technical Report 02/2008, School of Computer Sience, University of Magdeburg, 2008.

[29] C. H. P. Kim and K. Czarnecki. Synchronizing cardinality-based feature models and their specializations. In *ECMDA-FA'05: Proceedings of the First European Conference on Model Driven Architecture - Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*. Springer, 2005.

[30] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, San Diego, CA, USA, 1985.

[31] F. Loesch and E. Ploedereder. Restructuring variability in software product lines using concept analysis of product configurations. In *CSMR'07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 159–170, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

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

[33] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*. Springer, 1997.

[34] D. B. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 1999.

[35] I. Şavga, M. Rudolf, S. Götz, and U. Aßmann. Practical refactoring-based framework upgrade. In *GPCE'08: Proceedings of the Seventh International Conference on Generative Programming and Component Engineering*, Nashville, Tennessee, USA, October 2008. ACM.

[36] R. V. D. Staeten, V. Jonckers, and T. Mens. A formal approach to model refactoring and model refinement. *Software and Systems Modeling (SoSyM)*, 6(2):139–162, June 2007.

[37] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel. Refactoring UML models. In *UML'01: Proceedings of the Fourth International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148. Springer, 2001.

[38] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In C. Consel and J. L. Lawall, editors, *GPCE'07: Proceedings of the Sixth International Conference on Generative Programming and Component Engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

[39] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001.

[40] S. Trujillo, D. Batory, and O. Díaz. Feature refactoring a multi-representation program into a product line. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *GPCE'06: Proceedings of the Fifth International Conference on Generative Programming and Component Engineering*, pages 191–200, New York, NY, USA, 2006. ACM.

[41] N. van Eetvelde. *A graph transformation approach to refactoring*. PhD thesis, Antwerp, April 2007.

[42] M. Werner. *Facilitating Schema Evolution With Automatic Program Transformation*. PhD thesis, Northeastern University, July 1999.