Enabling and Using the UML for Model Driven Refactoring

Pieter Van Gorp*, Hans Stenten*, Tom Mens† and Serge Demeyer*

*Lab on Re-Engineering
University of Antwerp, Belgium
Email: {pieter.vangorp,hans.stenten,serge.demeyer}@ua.ac.be
http://win-www.uia.ac.be/u/lore/

†Programming Technology Lab
Vrije Universiteit Brussel, Belgium
tom.mens@vub.ac.be
http://prog.vub.ac.be/

***Abstract*— There is a historical gap between UML tools and refactoring tools. The former tools are designed to produce analysis and design *models* whereas the latter are designed to manipulate program *code*. MDA tool vendors aim to bridge this gap by regenerating program code from evolving UML OOAD models and vice versa. In this position paper, we describe the problems and a novel solution for implementing traditional OO source refactorings into the infrastructure from UML tools. After discussing the practical feasibility of advanced UML refactoring applications, we present our running tool prototype and round up with an overview of our future work on model driven refactoring.**

***Index Terms*— UML, Refactoring, Code Generation, MDA, Metamodeling, OCL, SDM, FAMIX, XMI, XUpdate, Tools**

I. CONTEXT

A *refactoring* is defined as a "behavior preserving program transformation" [Opd92]. Refactorings for OO software are based on the redistribution of classes, variables and methods across the class hierarchy in order to facilitate future adaptations and extensions. Especially with the recent trend in agile software development, refactoring receives widespread attention and consequently many integrated development environments (IDEs) are incorporating refactoring features into their tools.

II. PROBLEM

Unfortunately, the standard UML metamodel is inadequate for maintaining the consistency between a design model and the corresponding code. This is mainly because the UML metamodel considers the whole method body as implementation specific [OMG01]. Therefore, typical MDA tools consider method bodies as "protected areas", which must be supplied manually and are preserved over code (re)generation [Obj03]. It is within such protected areas that inconsistencies will be introduced when a UML model gets refactored. Consider the simple *Rename Class* refactoring: class names may be used within protected areas in type declarations, type casts and exceptions, and these will not be updated accordingly.

This problem can be bypassed pragmatically by using a language specific syntax tree for the protected areas, but this has several disadvantages. MDA tools become less maintainable as they need to implement the same refactorings for different programming language syntax trees. Similarly, developers lack a language-independent metamodel for specifying user-defined code smells. Lastly, parsing, transforming and regenerating code for the full syntax tree of industrial-strength OO languages may consume more time and memory than is required for executing refactorings.

As Don Roberts explains in his Ph.D. thesis, building a refactoring tool involves more than implementing the program transformations [Rob99]. The tool should also be able to check the invariants, pre- and postconditions of a (sequence of) refactoring(s) to ensure behavior preservation. One may also want to trigger refactorings based on the presence of the bad code smells they can solve. As these checks tend to make use of information inside the method-body, this calls even more for an extension of the UML metamodel.

III. SOLUTION: EXTENDING THE UML METAMODEL

GrammyUML is our initial proposal for a set of minimal and backward compatible extensions to the UML metamodel that enables UML tool vendors to keep their repository in sync with the code while applying common refactorings. We've already demonstrated how the pre- and postconditions, invariants and bad code smell specifications of the *Pull Up Method* and *Extract Method* refactorings can be expressed as OCL constraints on this UML metamodel extension [GSMD03]. These initial examples represent design level and implementation level refactorings respectively.

The goal of our research on *GrammyUML* is not to define the ultimate UML refactoring extension, but rather to provide concrete suggestions on how such an extension might look like. We learned that any UML extension for refactoring should

1) relate a method body to its contained statements and
2) leverage the profile mechanism to model language-specific features such as conditionals, exceptions, type casts, ... as cross-language abstractions that express strictly the information needed by the refactoring catalog in use.

Fig. 1. The language-independent core of *GrammyUML*, a minimal and backward compatible extension of the UML 1.4 metamodel.

Figure 1 displays the core of the *GrammyUML* metamodel that we updated based on expert feedback and further analysis of today's OO languages. We have reused UML 1.4's `Action` entity, which can contain `Arguments`. However, where an `Action` can apply to more than one target (e.g. a collapsed group of objects with the same type in a sequence diagram), our `SingleTargetAction` applies to one target only. `CallAction` is no subclass of `SingleTargetAction` due to our backward compatibility design criterium.

We are building a UML profile for refactoring on top of this core and validate it's fitness by implementing sample refactorings. We have not yet found statements in Java or Smalltalk that could not be modeled by (compositions of) stereotypes for `SingleTargetAction`.

Based on our experience with FAMIX, we use the `AccessAction` and `UpdateAction` stereotypes for `SingleTargetAction` [SDS99]. The `targetRefinement` of `SingleTargetActions` with such stereotypes can be a local variable, formal parameter or an attribute. In addition, we propose a similar `ClassAction` stereotype for type-casts (and other possible primitive source statements referring a class name).

IV. Application: Leveraging *GrammyUML*'s Refactoring Assets

OCL empowered CASE tools could detect where to apply a refactoring and compose sequences of primitive refactorings if they would support a metamodel similar to *GrammyUML*. The advantage of OCL is it's general purposeness (making it attractive to implement for CASE tool vendors and to learn for developers) and language-independence (making the code smell specifications reusable across common OO languages such as C++, Java, C#, Smalltalk).

In practice, implementing composite refactorings based on the OCL contracts is hindered by the lack of a backward chaining logic inference engine in current OCL evaluators. A logic inference engine would allow us to declaratively find refactorings whose postconditions make true unsatisfied preconditions of refactorings that should be executed (e.g. due to a bad code smell occurrence).

V. Validation

We are validating the practical feasibility of our theory by implementing the discussed refactorings in the Fujaba tool [FNTZ98]. This tool generates executable Java code from

Story Driven Modeling (SDM) diagrams. Figure 2 shows how SDM can be used to graphically model constraints on a metamodel (which can be evaluated on parsed model instances). In addition, SDM allows us to express the actual model *transformations* from refactorings even more intuitively (the `execute` story). The resulting refactoring can be invoked from UML diagrams: figure 3 for example shows our proof-of-concept on a UML class diagram.

VI. Future Work

In our future work on Model Driven Refactoring with *GrammyUML* we may cover additional refactorings, additional formalisms and additional tool validation.

We are working on both additional primitive OO refactorings and high level composed refactorings supporting design and architecture evolution.

As explained in section V, we have not yet been able to implement our long term tool setup where there is a clear separation between an applicability checker, a transformation engine and a well formedness checker. As figure 4 illustrates, we originally believed that OCL would be the appropriate formalism to use the former and the latter. However, due to the practical inpopularity of OCL (both in use by developers as in tool support) we are evaluating other formalisms (such as SDM) as well.

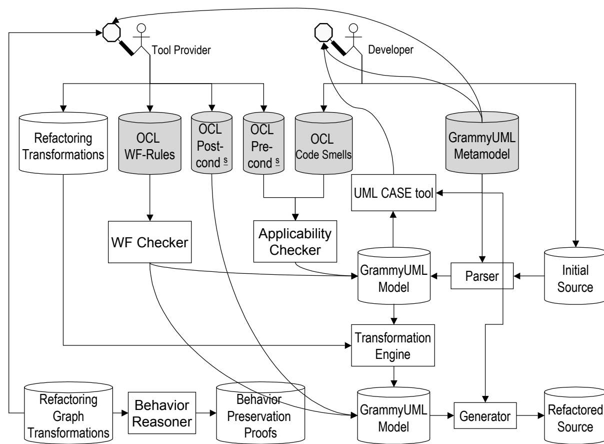

Fig. 4. To implement refactorings, *GrammyUML* models have to be parsed from – and generated to – program sources. Both the metamodel and instances hereof can then be visualized in UML tools. Developers can select the model elements that should be refactored by selecting them in a UML diagram or by specifying OCL bad code smell queries on the metamodel. Once implemented, runtime conformance can be checked by the OCL well-formedness rules and postconditions.

We are also evaluating how the emerging XQuery and XUpdate standards can be used to implement refactorings on XMI representing *GrammyUML* models. Our goal is to compare our graphical (SDM), in-memory implementation (in Fujaba) with a textual (XML), database implementation mainly in terms of expressiveness and scalability.

From a tool builder's perspective, we are investigating how the language independence of *GrammyUML* can be extended to the code regenerator architecture of a refactoring tool. Our goal is to build a tool prototype where code regeneration is nót handled by a hand-crafted and file-format specific visitor that follows fixed code formatting rules. Instead, parsing a small metamodel extension of today's *GrammyUML* should enable code regeneration as a service from the refactoring tool. In addition, this service would maximally preserve original code formatting. Possible validation cases include refactoring C++ code and Java middleware components. Our approach would preserve not only the C++ programmer's code conventions concerning white-spaces and newlines, but would also preserve hand-written forward declarations across `cpp` and header files (which are often designed as API documentation). Java middleware component models (such as EJB, JDO, Hibernate, ...) on the other hand rely more and more on XML deployment descriptors to separate database information (SQL types, transaction and caching attributes, ...) from the Java code [Gor02]. For performance optimizations, there can even be more than one descriptor per component [Jew01]. MDA tool vendors are currently providing dedicated parsers ánd code generation templates for round-trip engineering between such XML files and their UML models. We would validate to what degree the need for such templates could be reduced by our new code regenerator architecture where all textual information that is not contained in it's metamodel is automatically preserved.

References

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6th *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

[Gor02] Pieter Van Gorp. Vergelijkende studie van J2EE applicatieservers. Master's thesis, Universiteit Antwerpen, 2002.

[GSMD03] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Formal UML support for the semi-automatic application of object-oriented refactorings. Technical report, University of Antwerp, 2003. Submitted to the 6th International Conference on « UML » – The Unified Modeling Language. Available at: http://win-www.ruca.ua.ac.be/u/lore/refactoringProject/index.php?title=Publications.

[Jew01] Tyler Jewell. Unlocking the true power of entity EJBs. *ONJava*, 12 2001. Available at: http://www.onjava.com/pub/a/onjava/2001/12/19/eejbs.html.

[Obj03] Interactive Objects. ArcStyler. URL: http://www.arcstyler.com/, March 2003.

[OMG01] Object Management Group. Unified Modeling Language (UML), version 1.4, 09 2001.

[Opd92] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Rob99] Don Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[SDS99] Sander Tichelaar Serge Demeyer and Patrick Steyaert. Famix 2.0 – the famoos information exchange model. URL: http://www.iam.unibe.ch/ famoos/FAMIX/, 09 1999.

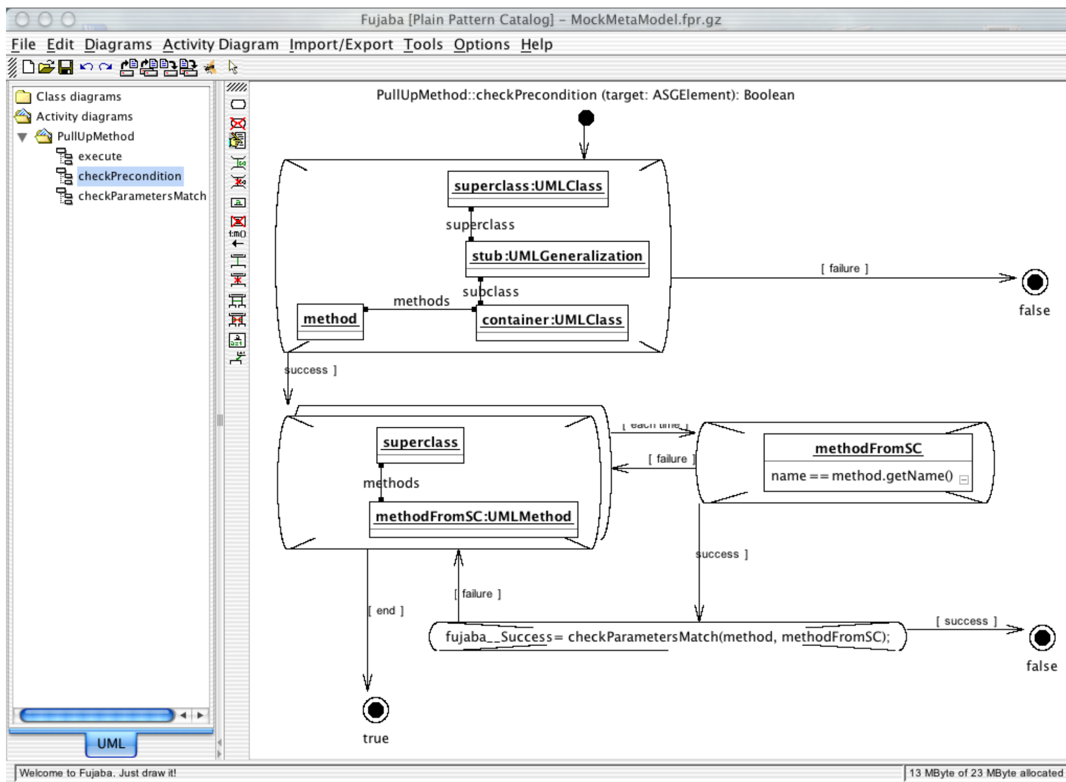

Fig. 2. Fujaba screenshot of our executable SDM specification for the precondition of the Pull Up Method refactoring. We check if the signature is not yet present in the parent of the containing class. Note that this proof-of-concept constraint does not yet guarantee correct refactoring behavior. For example, additional checks are required for the occurrence of (possibly semantically equivalent) methods with the same signature in siblings from the containing class.

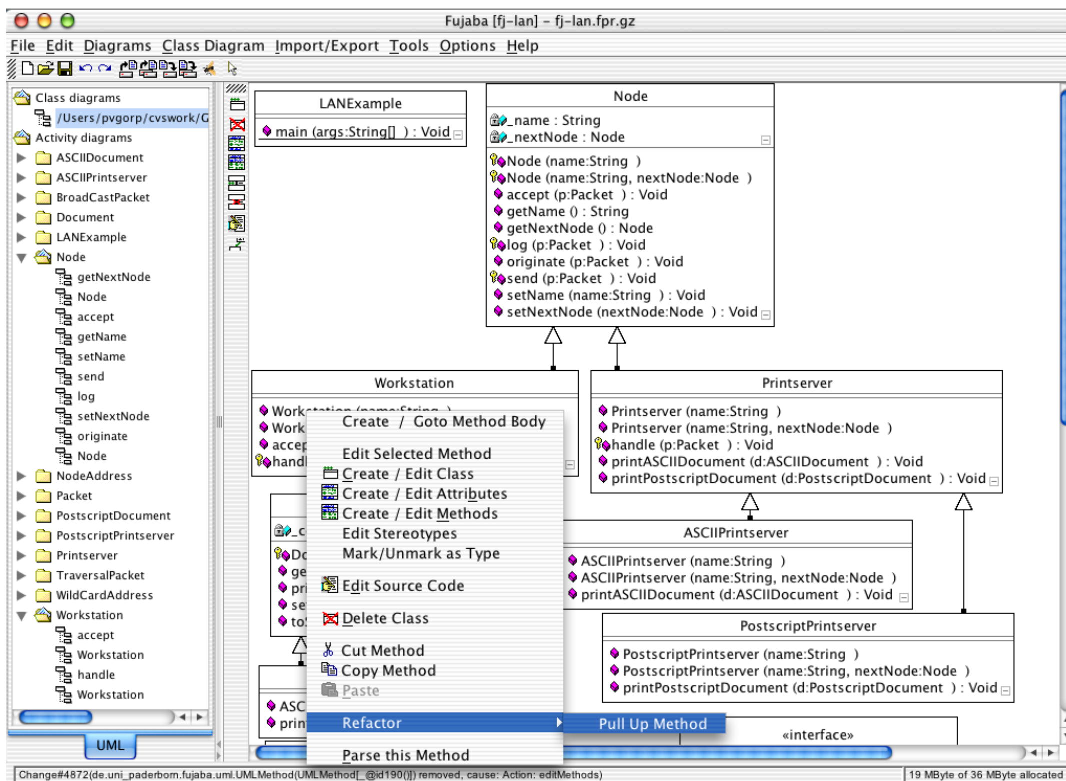

Fig. 3. Fujaba screenshot where the user has right-clicked on the `handle` method from `Workstation` in order to execute the Pull Up Method refactoring.