Towards automating source-consistent UML Refactorings

Pieter Van Gorp[1], Hans Stenten[1], Tom Mens[2], Serge Demeyer[1]

[1] Lab on Re-Engineering
University of Antwerp, Belgium
{pieter.vangorp,hans.stenten,serge.demeyer}@ua.ac.be
http://win-www.uia.ac.be/u/lore/
[2] Service de Génie Logiciel
Université de Mons-Hainaut, Belgium
tom.mens@umh.ac.be

Abstract. With the increased interest in refactoring, UML tool vendors seek ways to support software developers in applying a (sequence of) refactoring(s). The problem with such tools is that the UML metamodel – on which their repository is based – is inadequate to maintain the consistency between the model and the code while one of them gets refactored. Therefore, we propose a set of minimal extensions to the UML metamodel, which is sufficient to reason about refactoring for all common OO languages. For instance, by specifying pre- and postconditions in OCL, we are able to compose primitive refactorings, verify preservation of program behavior, and trigger refactorings based on code smells. This way, we provide future MDA tools with the ability to improve existing UML designs, yet keeping them in synch with the underlying code base.

1 Introduction

An intrinsic property of software in a real-world environment is its need to evolve. As the software is enhanced, modified and adapted to new requirements, the code becomes more and more complex and the original design slowly erodes. Therefore, it is not surprising that the major part of the total software development cost is devoted to software maintenance [1,2,3,4]. What may seem surprising though, is that better software development methods and tools do not reduce but *enlarge* the maintenance cost [5]. This is explained by the observation that better methods and tools are mainly used to accommodate even more requirements, thereby increasing the rate of change and consequently the effects of erosion.

To cope with design erosion there is a need for techniques that reduce software complexity by incrementally improving the internal software structure. The research domain that addresses this problem is referred to as *restructuring* [6,7]. In the domain of object-oriented software development however, the term *refactoring* is used instead, where it is defined as "behavior preserving program transformation" [8,9]. Refactorings 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 [10,11].

However, building a practical refactoring tool is not as straightforward as one might think, because refactorings are typically expressed at the design level yet must be aware of all the detailed code-level issues. For example, a software designer might rename a class to improve readability and traceability, but the refactoring tool must make sure that this does not cause any name collisions and that all references are updated correctly (including all special references like type-casts, exceptions, etc.). As another example, a software designer might pull up a method so that it can be reused by more subclasses, however the refactoring tool must verify that variable references inside the method body allow this and remove all other copies of the same method (hence tools must decide on the equality of methods). As a final example, a software designer might decide to reuse a piece of code by extracting it in a separate method, however the refactoring tool must ensure that the variables referred to are turned into method parameters or function results.

Since software designers think about refactorings at the design level, it is quite natural to exploit the UML and its unified metamodel for expressing refactorings. Indeed, the advantage of refactoring at the UML level is that software designers don't have to worry about the syntaxes of all the possible programming languages (C++, Java, C#, Smalltalk, ...), including all dialects and versions. Tool vendors on the other hand, can exploit the UML to implement behavior preserving code transformations with maximal reuse across implementation languages. With the new Model Driven Architecture (MDA) standard, this becomes especially relevant, since the UML is designed to serve as a basis for code (re)generation [12].

Unfortunately, the standard UML metamodel is inadequate for maintaining the consistency between a design model and the corresponding program code. This is mainly because the UML metamodel considers the whole method body as implementation specific [13]. Therefore, typical UML tools consider method bodies as "protected areas", which must be supplied manually and are preserved over code (re)generation [14,15]. It is within such "protected areas" that inconsistencies will be introduced when a UML model gets refactored. Consider the simple *Rename Class* refactoring: in the UML metamodel this is an almost atomic operation, however class names may be used within protected areas in type declarations, type casts and exceptions, and these will not be updated accordingly. More elaborate refactorings (e.g. *Pull Up Method*) will cause more severe problems, because necessary preconditions –the accessing of attributes– cannot be verified. Finally, some refactorings (e.g. the frequently used *Extract Method*) are close to impossible to express, because they require a precise model of the statement list in a method body.

Nevertheless, it is clear that the UML metamodel is nearly sufficient to express the effect of refactorings on source code. Therefore, this paper makes a "proof by construction" to show that it is possible to solve the discussed problem: we define a small extension to the UML metamodel which allows to verify the pre- and postconditions of two representative refactorings (i.e. Pull Up Method and Extract Method). The goal of this experiment is not to define the ultimate UML refactoring extension, but rather to provide concrete suggestions on how such an extension could be realized. Moreover, we show that such an extension can be applied in an OCL empowered CASE tool to compose sequences of primitive refactorings and to detect where a given refactoring should be applied.

This paper is structured as follows. Section 2 introduces some design criteria that should be met by a refactoring extension for UML, then makes a concrete suggestion on how such an extension could be realized, and afterwards discusses some advantages, drawbacks and alternatives. In section 3 we cover two sample refactorings to illustrate how refactoring pre- and postconditions can be specified in OCL. This enables the run-time verification of behavior preservation of a refactoring engine. Section 4 illustrates two applications of automated UML refactorings, namely composition of refactorings, and detection of code smells. After giving an overview of some related work on refactoring in general and in a UML context (section 5), the paper presents future work (section 6) and comes to a conclusion (section 7).

2 A UML Metamodel Extension for source-consistent Refactoring

This paper argues that the current UML 1.4 metamodel is inadequate for maintaining the consistency between a refactored design model and the corresponding program code. Therefore, it makes a concrete suggestion for extending the metamodel. However, while the UML has been designed as an extensible modelling language, it is clear that touching the metamodel itself may have a great impact on the supporting tool suites, and hence it should be designed carefully.

We focus on UML 1.4 for several reasons. From a tool compatibility and understandability perspective, we have selected UML 1.4 as it was the latest official version available for the development of this paper. More fundamentally and timelessly, we target UML 1.4 as it represents a family of UML versions without a precise *Action Semantics* package. While this package fills in some of the gaps we identified in the UML 1.4 metamodel, it is not necessarily optimal for our refactoring purposes [16]. We want to investigate whether a UML extension with the sole purpose of supporting refactoring has advantages over a fully executable UML, such as UML 1.5. This is especially relevant as not all UML 2.0 proposals include the *Action Semantics* package [17,18]. Within the context of MDA, we are hence investigating whether or not the the *Action Semantics* package is at the desirable abstraction level for refactoring.

2.1 Criteria

Below is a list of design criteria that we feel should be met by any UML version aimed at supporting refactorings. We will use these criteria to evaluate both our UML 1.4 extension, called *GrammyUML*, and the recent UML version 1.5. As such, we will illustrate the difficult trade-offs involved in designing a UML metamodel (extension) .

Integration with Code Smell Detectors Within the refactoring community, it is common to use the concept of “code smells” to indicate problematic code fragments that should be refactored [9]. In order to automatically suggest (and possibly execute) refactorings based on a set of detected code smells, a UML tool should present a metamodel that is adequate for describing such code smells and exchanging them with a detector [19,20].

Consistency Maintenance between Model and Code Since source code is considered as the primary software artifact in today's refactoring tools, refactoring implementations are by no means integrated with the repository of UML tools. MDA tools provide all the infrastructure needed to solve this consistency problem: a configurable parser, model transformator and code generator [14,15]. Still, source-consistent refactoring on UML models is only feasible if the UML metamodel is sufficiently expressive to capture the effects of refactorings on the underlying sources. Otherwise, the protected areas generated by the MDA tools will contain invalid references after refactoring the UML models.

Minimal Additions The UML specification describes two extension mechanisms, one being the "lightweight" profile mechanism, the other being the "heavyweight" MOF approach. The profile mechanism is restricted to adding attributes and constraints to *existing* model elements, whereas MOF allows the addition of *new* model elements. The more lightweight the extension, the easier it is for existing tools to accommodate them into their repository, hence its better to keep the extension as minimal as possible.

Maximal Effect By careful selection of the places where the metamodel is extended, it is possible that a minimal extension has maximal effect, i.e. that it is applicable to more than one diagram type. For instance, if an extension is applied in the *Behavioral Elements.Common Behavior* package, its immediately available to all diagram types, while an extension in the *Behavioral Elements.Collaborations* package is only applicable to sequence and collaboration diagrams.

Backward Compatibility When extending or reusing an existing model element, changes to the semantics should be avoided. Such conservative extensions guarantee that all existing UML models remain valid.

For all Common OO Languages One of the key benefits of the UML metamodel is that it abstracts away language syntax without loosing the basic OO constructs (classes, methods and attributes, ...). When extending this core, we should balance between achieving enough precision for refactoring and maintaining language independence.

2.2 *GrammyUML* – Refining the Method Body

Heavyweight Extensions As demonstrated in the introduction, the UML 1.4 metamodel represents the body of a method as an unspecified *ProcedureExpression* datatype which makes it inadequate for refactoring. To cope with this problem, we propose 8 additive and language-independent extensions to the UML 1.4 metamodel which form the foundation of *GrammyUML*:

1. relate a *Method* to its contained statements, i.e. to *ActionSequence*,
2. add *LocalVariable* as a specialisation of *ModelElement*,
3. relate a *LocalVariable* to its type, i.e. to *Classifier*,
4. relate a *LocalVariable* to its surrounding scope, i.e. to *ActionSequence*,
5. relate an *Action* to its actual arguments, i.e. to *Argument*,

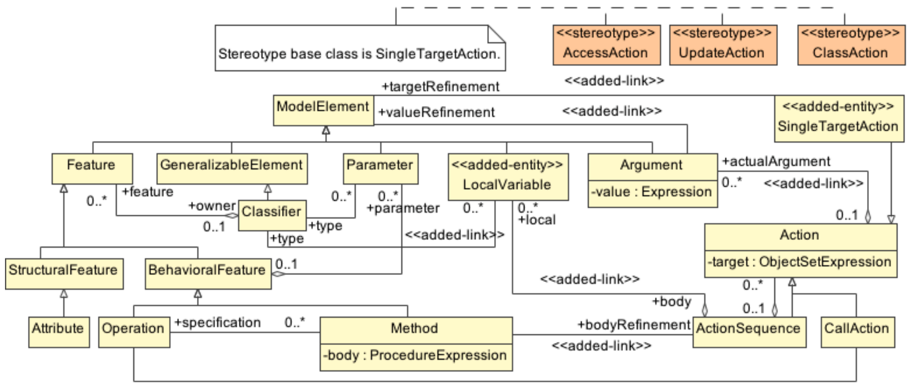

Fig. 1. GrammyUML metamodel – simple, yet adequate for refactoring.

6. refine the *value* attribute of *Argument* to the *valueRefinement* association-end of type *ModelElement*,
7. add *SingleTargetAction* as a specialization of *Action*,
8. refine the *target* attribute of *SingleTargetAction* to the *targetRefinement* association-end of type *ModelElement*.

The first four extensions are applied to the *Foundation.Core* package. They allow to model the statements in a method body and the use of typed local variables in a given scope. The other extensions are applied to the *Behavioral Elements.Common Behavior* package. The *valueRefinement* association-end of *Argument* to *ModelElement* allows to specify which *LocalVariable*, *Parameter* or *Attribute* is passed as actual argument to an *Action*. Note that "return values used as as arguments" and "cascaded method calls" can be modeled using implicit local variables.

The *Action*s representing the method body can symbolize any language specific construct. By definition, extensions for refining such constructs do not belong in the language-independent foundation of the UML metamodel. However, the plain *Action* concept of UML 1.4 is not suitable for representing primitive programming statements. Such statements mostly apply to one target whereas the *target* attribute of *Action* is of type *ObjectSetExpression*. The last two extensions rectify this problem.

Lightweight Extensions From our experience with the FAMIX and GRAMMY metamodels we know that the notion of access-, call- and update-behavior augmented with the notion of type-casting is a stable basis to reason about refactoring consistency [21,22]. We easily integrate access-, and update-behavior in our metamodel extension by defining the *AccessAction* and *UpdateAction* stereotypes on *SingleTargetAction*. Similarly, we model type-checks and type-casts with a *ClassAction* stereotype. We can reason about call-behavior using the *CallAction* that is part of UML 1.4.

Well-Formedness Rules The UML metamodel has well-formedness rules (WFRs) to express invariant constraints on the metamodel elements, e.g. there cannot be two methods with the same signature in a class. As *GrammyUML* is an additive extension, all existing WFRs of the UML 1.4 metamodel remain valid after introducing the extra WFRs. Although it is straightforward to express the new refactoring WFRs in OCL, we present them in natural language:

1. The *targetRefinement* of a *SingleTargetAction* stereotyped as *AccessAction* or *UpdateAction* must be a *LocalVariable*, a *Parameter* or an *Attribute*.
2. The *valueRefinement* of an *Argument* must specify a *LocalVariable*, a *Parameter* or an *Attribute*.
3. The *targetRefinement* of a *SingleTargetAction* stereotyped as *AccessAction* or *UpdateAction* cannot be an *Attribute* defined in a subclass.
4. The *targetRefinement* of a *SingleTargetAction* cannot be a *LocalVariable* or a *Parameter* defined in another method.

2.3 UML 1.5 metamodel: Action Semantics

The *Action Semantics* package consists of a set of programming language abstractions for modeling the creation and destruction of objects, the flow of data between components and their control flow (conditionals, loops, constructs for concurrency and asynchronous behavior, ...) Most of the GrammyUML extensions can be found within this package that has been included in UML 1.5.

2.4 Evaluation

In this section we will briefly evaluate our refactoring focussed extension and the recent UML 1.5 version.

UML 1.5 contains sufficient information to trace the effect of most refactorings inside method bodies and specify OCL code smells that make use of statement-level information. Even stronger, UML 1.5 is adequate to express control- and data-flow in its *Action Semantics* specification. *GrammyUML* does not contain such detailed information. Conversely, type-checks and type-casts can only modeled with *GrammyUML*'s metamodel. Recall from the *Rename Class* example in section 1 that this can lead to incorrect refactoring behavior.

Figure 2 illustrates the difference in complexity between *GrammyUML* and UML 1.5 by showing the entities needed to reason about the local variables used in program statements.

In UML 1.5 it was decided to use heavyweight extensions, adding seperate entities like *ReadAttributeAction* in seperate packages like *Variable Actions*. The lack of a common supertype (or stereotype *baseClass*) for these similar types makes it very hard to make extensions like a type-check construct without adding even more entities and packages.

Lastly, *GrammyUML* is a conservative extension of UML 1.4. UML 1.5 isn't backward compatible with UML 1.4 since many entities are renamed (e.g. from *ActionSequence* to *GroupAction*) or deleted from the model (e.g. *Argument*) and many others are added with the semantics which was formerly covered by other entities.

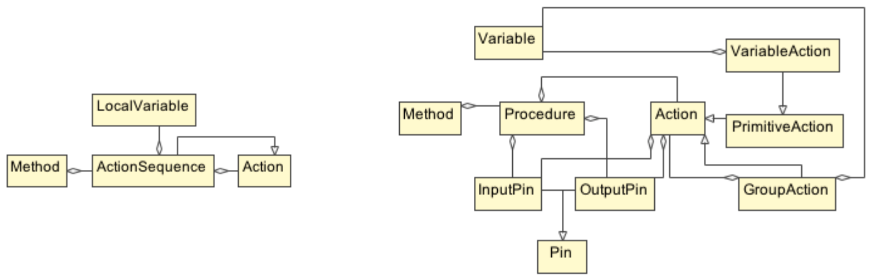

Fig. 2. Left: *GrammyUML* metamodel fragment. Right: UML 1.5 metamodel fragment. Although the control flow constructs and the component *pins* may be required for control and data flow analysis, *GrammyUML* shows that a refactoring metamodel can have a higher level of abstraction.

3 Automating UML Refactorings based on Refactoring Contracts

In this section we show that the extensions from subsection 2.2 are sufficient to reason about refactoring for all common OO languages. We first introduce the concept of *refactoring contracts*, a set of constraints that describe the effects of refactorings. Next, we use OCL to describe the refactoring contracts of 2 sample refactorings of realistic complexity: Extract Method and Pull Up Method. Most OCL constraints use user-defined properties for the sake of readability but, due to space considerations, we have left them out. The reader can make them as an exercise and / or contact the authors.

3.1 Refactoring Contracts

A refactoring contract consists of three sets of constraints per refactoring: (i) a *precondition* describes the model restrictions that need to be satisfied before applying the refactoring, (ii) a *postcondition* states what model properties are guaranteed by the refactoring, and (iii) the *code smells* describe the problematic model constructs that can be improved by the refactoring.

By carefully selecting the pre- and postconditions of a contract, certain behavior properties can be guaranteed. In this paper, we use refactoring contracts for the preservation of *access*-, *update*- and *call-behavior* [22].

3.2 Extract Method

The Extract Method refactoring turns a sequence of statements from a method *oldM* in class *C* into a new method *m* in *C*. The aim of this refactoring is to remove code that is duplicated across multiple methods of the same class and / or raise the intentionality of a long method [9].

Precondition The method *m* may not give a name conflict in the inheritance hierarchy of *C*.

```
context Method
  def:
  let pre_extract_method(name : String) : Boolean =
    not self.owner.inheritanceTree->exists(class |
      class.getSignatures->exists(method |
        method.name = name
      )
    )
```

Postcondition (i) *m* exists in *C*, (ii) each local variable, attribute or formal parameter used by the statements is passed to *m*, and (iii) each occurence of the statements is replaced by a method call of *m*.

The OCL for (i) is similar to the above precondition specification while (ii) and (iii) require more complex model traversals. We present the second part of the postcondition in OCL below as it nicely illustrates how method body information can be collected.

```
context Method
  def:
  let pass_variables_as_parameters(
                   extrStatements: ActionSequence,
                   oldM: Method,
                   m : Method) : Boolean =
    extrStatements.action->select(oclIsTypeOf(AccessAction) or
                                  oclIsTypeOf(UpdateAction))
    .oclAsType(SingleTargetAction).targetRefinement->select(
     tr |
       ( oclIsTypeOf(LocalVariable) and
         not (
           ( tr.oclAsType(LocalVariable)
               .isDeclaredIn(oldM.bodyRefinement
                                 .without(extrStatements)) and
             not tr.oclAsType(LocalVariable)
                   .isUsedIn(oldM.bodyRefinement
                                 .without(extrStatements))
           ) or
           tr.oclAsType(LocalVariable)
             .isDeclaredInChildOf(oldM.bodyRefinement)
         )
       ) or
       oclIsTypeOf(Parameter)
    )->forAll(tr |
      m.parameter->exists(parameter |
        parameter.name = tr.name and (
          parameter.type = tr.oclAsType(LocalVariable)
                               .type or
          parameter.type = tr.oclAsType(Parameter).type
        )
      )
    ) and
    extrStatements.action->select(ActionSequence)->forAll(
    childSeq |
      pass_variables_as_parameters(childSeq, oldM, m)
    )
```

Code Smells One of the goals of the Extract Method refactoring is to handle the duplicated code code smell as described in [9].

In this example we show an OCL expression checking two sequences of statements for equality, we use this because in this form the expression is also used in the post-condition (iii) of the Extract Method refactoring. For the sake of completeness we have to mention that this interpretation of duplication isn't the only one [23]. In the example two sequences are given together with two integers, denoting the entries in each sequence that are compared with each other.

```
context Method
  def:
  let check_statement_match(src : Sequence(Action),
                              i : Integer,
                              extract : Sequence(Action),
                              j : Integer) : Boolean =
    if j >= extract->size then
      true
    else
      if src->at(i) = extract->at(j) then
        check_statement_match(src, i+1, extract, j+1)
      else
        false
      endif
    endif
  -- shorthand for client call
  let check_statement_match(src : Sequence(Action),
                           extract : Sequence(Action)) : Boolean =
    check_statement_match(src, 0, extract, 0)
```

3.3 Pull Up Method

The Pull Up Method refactoring moves a method *m* with implementation *MD* of a class *C* into its superclass *SC*. After this first move, all methods carrying the same signature in *C*'s sibling classes can be removed. The aim of this refactoring is to centralize functionality in a common super class.

Precondition (i) *SC* must exist, i.e. *C* may not be a root class, (ii) the signature *m* may not already exist in superclass *SC*, (iii) the implementation of *m* shouldn't refer to any variables defined in *C*, and (iv) the implementation of *m* shouldn't refer to a method not accessible in *SC*.

```
context Method
  def:
  let pre_pull_up_method : Boolean =
    not self.owner.superClass->notEmpty and -- 1
    not self.owner.superClass.getSignatures->exists(method |
      method.signatureCollision(self)) and -- 2
    not self.usesLocalAttributes and --3
    not self.usesLocalMethods -- 4
```

Postcondition (i) The signature *m* exists in superclass *SC*, (ii) this signature is implemented by a body that is equivalent to *m*, (iii) methods with *m*'s signature and an equivalent body are removed in all direct subclasses of *SC*, and (iv holds implicit) methods with *m*'s signature and a different body are preserved in *SC*'s direct subclasses

```
context Method
  def:
  let post_pull_up_method(oldSelf : Method) : Boolean =
    self.owner = oldSelf.owner.superClass and -- 1
    self.signatureCollision(oldSelf) and -- 1
    check_statement_match(self.bodyRefinement.action->asSequence,
          oldSelf.bodyRefinement.action->asSequence) and -- 2
    not oldSelf.owner.allSiblings.getMethods->exists(m |
      m.signatureCollision(self) and
      check_statement_match(self.bodyRefinement.action->asSequence,
            m.bodyRefinement.action->asSequence)) -- 3
```

Code Smells We leave the specification of code smells for Pull Up Method as an exercise to the reader. The table at the end of Fowler's book can be used to relate code smells and refactorings [9].

3.4 Evaluation

We have shown that it is feasible to use OCL to describe refactoring contracts on *GrammyUML*'s metamodel. The OCL *let*-defined constraints can be integrated with MDA CASE tools in various ways. The obvious application would be to effectively use the proposed pre- and postconditions as OCL pre- and postcondition constraints on the UML operation that represents the refactoring under consideration, but this can be left open to CASE tool vendors.

4 Applications of Automated UML Refactoring

4.1 Compose primitive refactorings

Roberts analysed *composite refactorings* (i.e., sequences of more primitive refactorings), in more detail [24]. In the example on the next page, we illustrate how OCL refactoring contracts can be used to compose primitive refactorings.

```
class A extends SC {
        method m {
                X
                if (i == a) {
                        Y
                }
                Z
        }
}
```

```
class B extends SC {
        method m {
                X
                if (i == b) {
                        Y
                }
                Z
        }
}
```

Two classes *A* and *B* with a common superclass *SC* both have a method *m* with almost the same implementation. The method *m* could be pulled-up into the superclass. However the precondition of the *Pull Up* method refactoring as described in subsection 3.3 is violated. The two conditionals do not match, so the implementation of the methods isn't identical.

However if we consider other primitive refactorings as described by Opdyke and Roberts (*Add Method*) it is possible to find a sequence of refactorings that solve this mismatch and whose pre- and postconditions guarantee behavior preservation [8,24].

When we extract the conditional in both classes to a method with a name *testCondition* using Extract Method, the postcondition of this refactoring ensures that each occurence of the statements is replaced by a method call to *testCondition*. This results in two identical implementations of *m* in *A* and *B*.

Still, the Pull Up Method cannot yet be performed. The two implementations of *m* access a method (*testCondition*) not defined in the superclass *SC*, which violates the precondition of Pull Up Method. Combining the pre- and postconditions of the Extract Method refactoring, we know that *testCondition* does exist in *A* and *B*, but nowhere else in their inheritance tree. This satisfies the precondition of the *Add Method* refactoring applied on *SC* that requires that there is no method with the same signature as *testCondition* present in superclass *SC*. After applying this *Add Method* refactoring, its postcondition guarantees that *testCondition* exists in *SC*.

The postconditions of the Extract Method and *Add method* refactorings satisfy the precondition of Pull Up Method, hence this refactoring can be applied. Summarizing, by implementing the OCL refactoring contracts we proposed in this paper, UML CASE tool vendors can support automatic refactoring composition. Herefore, they need to implement a mechanism that – given a refactoring for which the code smell of its contract has fired but whose precondition is not (completely) satisfied – finds all refactorings whose postcondition can rectify this violated precondition. Note that this meets the state-of-the art in MDA, as this algorithm can only be implemented in a vendor specific manner due to the lack of a model transformation standard. Formal UML specification of model transformations is needed to deduce how a refactoring ensures the postcondition of its contract and vice versa.

4.2 Integrate with code smell detectors

It has already been demonstrated that OCL is adequate to specify design metrics on the UML metamodel [25]. In section 3 we showed how our extended metamodel enables us to collect metrics data from inside the method body and describe the code smells from Fowler in our refactoring contracts [9]. The next step is to agree on a treshold value that says when the code smells so bad that it needs to be refactored[3]. These tresholds can be stored in the context of the refactoring contracts for a specific language or application. Applying the logical "AND" from the OCL code smell with the OCL precondition of the same contract, to a *GrammyUML* model instance, results in a list of methods that

1. are *desired* to be transformed – as they exceed the treshold,
2. are *allowed* to be transformed while preserving their behavior – as the precondition of a refactoring that solves the problem is satisfied.

After applying the refactoring the model can be validated using the OCL postcondition from its contract.

5 Related Work

Fowler's refactorings are specified as source-to-source transformations [9]. The concept of model-driven refactoring has historically been investigated in the context of *design model* restructuring. Sunyé for instance explored how the integrity of class diagrams and statecharts can be maintained over refactoring [27]. The authors use the plain UML metamodel to describe refactoring pre- and postconditions in OCL. Due to the limitations of the UML 1.4 metamodel, however, they are unable to formalize the relation between the class (= structural) and statechart (= behavioral) diagrams. An extended metamodel like *GrammyUML* can solve this problem.

Van Der Straeten investigates how design diagram inconsistencies can be detected by specifying well-formnedness rules in description logic, a decidable fragment of first-order predicate logic [28]. Because there is no explicit formal relation between the UML diagrams and the program code, the model evolutions they investigate only apply to the analysis or design phase and do not fit into a process that makes use of round-trip engineering. By complementing their diagram consistency rules with our source-consistent refactoring contracts, this problem can be overcome.

In this paper, we have introduced the notion of *refactoring contracts* and have shown how such contracts can be formalized as OCL expressions on our UML metamodel extension. Opdyke was one of the first to describe primitive refactorings together with the *preconditions* that must be met to ensure that the transformation preserves program behavior [8]. Roberts extended this work by augmenting the refactoring definitions with *postconditions* and showing how refactorings can be composed [24]. Tourwé and Mens use logic metaprogramming to integrate *code smells* in an automatic refactoring tool [19]. Prolog-like logic rules are used to detect bad smells in the program code and to suggest refactorings that can be used to remove these smells. We have raised the

[3] The risk tresholds published by the Software Engineering Institute [26] can be used as initial values but they are likely to be aligned with each project's priorities.

abstraction level by using a language independent metamodel instead of the Smalltalk abstract syntax tree.

Evolution conflicts arise when parallel changes to the same (parts of the) program code or design by different developers give rise to inconsistencies. To detect and resolve these inconsistencies, the UML metamodel needs to be extended with versioning information or evolution information. [29,30] propose such an extension.

6 Future Work

In [31] we give an extensive overview of the different notions of behavior. We are actively researching the formalization of program behavior, going from *access-*, *update-* and *call-preservation* [22] to *language-preservation* [32]. Better notions of behavior may lead us to refinements of the initial contracts from section 3 while the *GrammyUML* metamodel can remain stable.

In this paper, we have applied *GrammyUML* to source code refactorings. However, *GrammyUML* may also greatly facilitate research on co-evolution between different UML diagrams as we have linked the method body from the *Foundation.Core* package with behavioral information from the *Behavioral Elements.Common Behavior* package. This relation is an elegant formal basis to link class diagrams with sequence diagrams.

Fowler's book presents 72 refactorings and many more have been found since its publication [9]. We will elaborate OCL refactoring contracts for more refactorings to validate the fitness of *GrammyUML*'s metamodel.

7 Conclusion

In this paper we have discussed an extension of the UML metamodel, which allows to express the pre- and postconditions for two representative refactorings (namely Pull Up Method and Extract Method). More precisely, the extension allows an OCL empowered CASE tool to (i) verify non-trivial pre- and postconditions, (ii) compose sequences of refactorings and (iii) use the OCL query engine to detect code smells. Of course, the experiment is limited in the sense that we did not yet prove that the UML extension works for all possible refactorings, nor did we verify whether this is the best possible way to extend the UML. Nevertheless, we have shown that it is both *feasible* and *desirable* to use the UML as a way to refactor designs independent of the underlying programming language. Both issues are crucially important for the coming generation of MDA tools, because they allow for a seamless integration between modelling and coding.

Figure 3 visualizes the role of a UML refactoring metamodel in future MDA tools. Both the metamodel and instances hereof can be visualized as UML class diagrams. UML metamodelers can describe custom code smells in OCL as we have demonstrated in section 3. Developers can parse their existing program sources to UML and use an applicability checker to automatically execute (sequences of) appropriate refactorings. The MDA tool preserves all occurrences of the metamodel entities and their surrounding text to regenerate the executable system. As a result, the extracted UML design has been improved without loosing its consistency with the source.

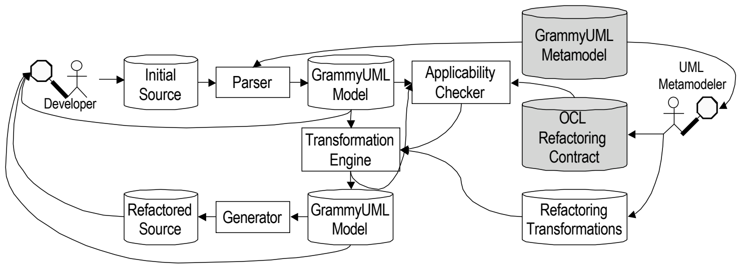

Fig. 3. GrammyUML Tool Context. The gray areas highlight the UML artifacts contributed by this paper.

References

1. B. P. Lientz and E. B. Swanson. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, 1980.
2. T. Guimaraes. Managing application program maintenance expenditure. *Comm. ACM*, 26(10):739–746, 1983.
3. J. T. Nosek and P. Palvia. Software maintenance management: changes in the last decade. *Journal of Software Maintenance and Evolution: Research and Practice*, 2:157–174, 1990.
4. M. Hanna. Maintenance burden begging for a remedy. *Datamation*, pages 53–63, 1993.
5. R. L. Glass. Maintenance: Less is not more. *IEEE Software*, July/August 1998.
6. Robert S. Arnold. *An introduction to software restructuring*, pages 228–269. IEEE Press, 1986.
7. William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, August 1991.
8. William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
9. Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, July 1999.
10. Don Wells and Laurie A. Williams, editors. *Proceedings of the Second XP Universe and First Agile Universe Conference (XP/Agile Universe 2002)*, volume 2418 of *Lecture Notes in Computer Science*. Springer, August 2002.
11. *Proceedings of the 4th International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2003)*. http://www.xp2003.org/, May 2003.
12. Object Management Group. Model Driven Architecture (MDA), document number ormsc/2001-07-01, July 2001.
13. Object Management Group. Unified Modeling Language (UML), version 1.4, September 2001.
14. Interactive Objects. ArcStyler. http://www.arcstyler.com/, March 2003.
15. Compuware. OptimalJ. http://www.compuware.com/products/optimalj/, March 2003.
16. Object Management Group. Unified Modeling Language (UML), version 1.5, March 2003.
17. U2 Partners. 3rd revised submission to OMG RFP ad/00-09-01: Unified Modeling Language: Infrastructure, version 2.0. http://www.u2-partners.org/, January 2003.

18. 2U Consortium. Unambiguous UML (2U) 3rd revised submission to UML 2 infrastructure RFP. http://www.2uworks.org/, January 2003.
19. Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2003.
20. Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, October 2002.
21. Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0 – the FAMOOS information exchange model. URL: http://www.iam.unibe.ch/ famoos/FAMIX/, 09 1999.
22. Tom Mens, Niels Van Eetvelde, Dirk Janssens, and Serge Demeyer. Formalising refactorings with graph transformations. *Fundamenta Informaticae*, 2003.
23. Filip van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques. In *International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) 2003*, 2003.
24. Don Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
25. Aline Lúcia Baroni. Formal definition of object-oriented design metrics. Master's thesis, Vrije Universiteit Brussel - Belgium and Faculty of Sciences in Collaboration with Ecole des Mines de Nantes - France and Universidade Nova de Lisboa - Portugal, 2002.
26. Software Engineering Institute. Cyclomatic complexity. http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html.
27. Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *UML 2001*, volume LNCS 2185 of *Lecture Notes of Computer Science*, pages 134–148. IRISA, Springer Verlag, 2001.
28. Ragnhild Van Der Straeten, Jocelyn Simmonds, and Tom Mens. Using description logic to maintain consistency between UML models. In *Proceedings of « UML » 2003 – The Unified Modeling Language*. Springer-Verlag, 2003.
29. Tom Mens, Carine Lucas, and Patrick Steyaert. Supporting disciplined reuse and evolution of uml models. In J. Bezivin and P.-A. Muller, editors, *Proceedings of « UML » '98 – The Unified Modeling Language*, volume 1618 of *Lecture Notes in Computer Science*, pages 378–392. Springer-Verlag, 1999. Mulhouse, France.
30. Tom Mens and Theo D'Hondt. Automating support for software evolution in uml. *Automated Software Engineering Journal*, 7(1):39–59, February 2000.
31. Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Language Descriptions, Tools and Applications (LDTA)*, 2002.
32. P. L. Bergstein. Maintenance of object-oriented systems during structural evolution. *Theory and Practice of Object Systems*, 3(3):185–212, 1991.