Model/Code Co-Refactoring: An MDE Approach

Jens von Pilgrim, Bastian Ulke, Andreas Thies, Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Hagen, Germany

***Abstract*—Model-driven engineering suggests that models are the primary artefacts of software development. This means that models may be refactored even after code has been generated from them, in which case the code must be changed to reflect the refactoring. However, as we show neither re-generating the code from the refactored model nor applying an equivalent refactoring to the generated code is sufficient to keep model and code in sync — rather, model and code need to be refactored jointly. To enable this, we investigate the technical requirements of model/code co-refactoring, and implement a model-driven solution that we evaluate using a set of open-source programs and their structural models. Results suggest that our approach is feasible.**

***Index Terms*—Model-driven engineering, refactoring, constraints**

I. INTRODUCTION

Model/code co-evolution [2, 4] is a classical problem of model-driven engineering (MDE): Every change to a model potentially leads to a change of the code generated from it, and if the generated code has been manually modified or enriched, generating it again bears the risk of losing or invalidating the added effort. Changing generated code on the other hand may require a corresponding adaptation of the model, unless one is willing to accept that the two get out of sync.

Model/code co-refactoring is a special case of model/code co-evolution in which changes are limited to ones that alter the structure of a system while preserving its behaviour [5]. Since model/code co-refactoring is usually targeted at non-functional requirements (such as readability or maintainability) rather than functional ones (which are usually paramount for the customer), chances are that it causes more problems than it solves, in which case it will not be done. Its added value will then be lost.

In this paper, we argue for and present a principled approach to model/code co-refactoring, that is, of jointly refactoring a model and the code generated from it in such a way that the code and model remain in sync. As we will point out, this requires tightly linked representations of both model and code, since changes in one representation may imply changes in the other representation which may in turn imply changes in the first. Our solution relies on the technique of constraint-based refactoring that has been used with good success in refactoring programs alone (see, e.g., [11, 17]), and extends it with cross-language constraints generated from the traces produced by the original model-to-code transformations. In addition, we show how significant parts of the specifications necessary for constraint-based model/code co-refactoring can be generated automatically from the metamodels of the languages involved in the co-refactoring (e.g., UML and Java), making our approach directly transferable to the co-refactoring of domain-specific languages (DSLs) and the language they are embedded in.

More specifically, we contribute the following:

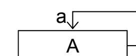
1. We show that model/code co-refactoring requires a tight integration of both model and code refactoring, including a feedback loop from co-refactored code to its model.
2. We show how independent specifications of model and code refactorings can be combined into model/code co-refactoring specifications by generating bridge constraints from the trace information produced by code generation.
3. We show how significant parts of the specifications needed for model/code co-refactoring can be generated automatically from the metamodels of the modelling language, the programming language, and the generation trace.
4. We demonstrate the feasibility of our approach by implementing it on top of the refactoring constraint language and framework REFACOLA [11], and by applying it to a large corpus of open source programs and their models.

The remainder of this paper is organized as follows. In Section II, we motivate our work by stating essential problems of model/code co-refactoring. Section III discusses related work and how it addresses these problems. Section IV provides a brief recap of constraint-based refactoring, on which our solution rests. In Section V, we detail how model/code co-refactoring can be expressed in terms of constraints. Section VI sketches how significant parts of the (declarative) specifications required for constraint-based model/code co-refactoring can be generated from the involved language specifications, if these are defined using a common metamodelling language such as Ecore. Section VII briefly describes our implementation, which is evaluated in Section VIII.

II. PROBLEM STATEMENT

To give the reader an impression of the problems of model/code co-refactoring, we use three simple examples involving UML class diagrams and Java.

EXAMPLE 1: The simple class diagram



978-1-4799-0215-6/13 © 2013 IEEE ASE 2013, Palo Alto, USA
New Ideas Track

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

expresses that class A has a unidirectional navigable association (aka reference) whose target is named a and has type A. From this diagram, the Java code

```
class A {
   private A a;
   public void setA(A a) { this.a = a; }
   public A getA() { return a; }
}
```

is typically generated. To enhance the generated code with implementation detail in such a way that the enhancements survive re-generation, a model engineer may add a subclass containing the enhancements, which is to be used instead of the generated class:

```
class AImpl extends A {
   private Object b;
   public void setB(Object b) { this.b = b; }
}
class C {
   public void foo() {
      AImpl aImpl = new AImpl();
      aImpl.setB(aImpl);
   }
}
```

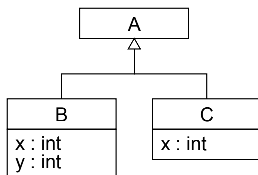
Now suppose that after reviewing system design, a modeller decides to rename the reference currently labelled "a" to "b". Re-generating the code then leads to (changes highlighted)

```
class A {
   private A b;
   public void setB(A b) { this.b = b; }
   public A getB() { return b; }
}
```

The program, including the manually added classes AImpl and C, still compiles, but its semantics is changed, since the call of setB(AImpl) in method C.foo() now binds to the inherited method A.setB(A), whose formal parameter type A is more specific to the actual parameter type AImpl than Object, the formal parameter type of the method to which it bound before (AImpl.setB(Object)). It follows that in presence of extensions to the generated code, re-generation from a refactored model is not sufficient for model/code co-refactoring, even if the extensions survive re-generation. ♦

The problem highlighted in Example 1 is that refactoring the model and regenerating the code may break an (implicit) contract between the originally generated code and its manual extension.

EXAMPLE 2: Consider the class diagram



and the code generated from it, manually enhanced directly in the generated class where the enhancement (highlighted) is tagged with the annotation @not_generated

```
public class A {}
public class B extends A {
   private int x;
   private int y;
   public int getX() { return x; }
   public void setX(int x) { this.x = x; }
   public int getY() { return y; }
   public void setY(int y) { this.y = y; }
   @not_generated
   public void init() { x = 10; y = 10; }
}
public class C extends A {
   private int x;
   public int getX() { return x; }
   public void setX(int x) { this.x = x; }
}
```

Now suppose that attribute x in the class diagram should be pulled up (from both B and C) to class A. When regenerating the code from the refactored model, it will no longer compile, since the (now pulled up) field x is no longer accessible to method B.init(). A sophisticated Java refactoring tool could fix this by increasing accessibility of x to protected as part of a model/code co-refactoring, but this change would be overridden once the model is re-generated. A better remedy would be to pull up init(), and with it y, to A as well, but the pulling up of y would mean a change of the class diagram that is not derivable as a necessary part of the refactoring from the class diagram alone. ♦

The lesson learnt from Example 2 is that a refactoring of a model may require changes in the code generated from it, which may in turn require changes of the model that would not be necessary had the refactoring been performed on the model alone. The following example adds an extra twist to this.

EXAMPLE 3: Consider the same model and code as that of Example 2 above, the only differences being that x is a constant having a different value in class B than in class C, and that init contains the (sole) assignment y = x. Now suppose that the modeller wants to move y to class A. As for pulling up x, on the Java side B.init must be moved with y, and so must B.x (but, because of the different values, not C.x!). However, this time, the change of the location of x need not only be reflected in the model also, it should cause the rejection of the refactoring, namely if the well-formedness of class diagrams requires that all members of a class, declared and inherited, must be named differently.[1] ♦

Here, the well-formedness rules of the modelling language prevent a model refactoring, but not because the model itself cannot be refactored, but because the model refactoring induces a code refactoring which in turn induces a change of the model that makes it ill-formed.

The above three examples show that mutual dependencies may exist between a model to be refactored and the code that is generated from it (including its manual extensions). In general, it is therefore neither sufficient to perform the refactoring on the model and then to re-generate the code, nor to perform the refactoring on the model and the generated code in parallel — rather, model and code must be treated as a unit that must be refactored together.

[1] Although UML does not define such a constraint, the Ecore metamodel [3], which is part of the Eclipse Modelling Framework widely used for code generation and which strongly resembles the class model of UML, has it.

III. Related Work

Model refactoring [15] is the natural extension of program refactoring to models. A comprehensive literature survey on model refactoring is given in [12] (and is not repeated here).

Constraint-based refactoring as forms the basis of this work was first proposed by Tip et al. [16, 17], who implemented a large array of type-related refactorings (such as Extract Interface, Generalize Declared Type, or Use Supertype Where Possible) with it. The approach was later extended by our group to refactoring accessibilities [10] and names [11]. Recently, we showed how constraint-based refactoring can be extended to models, with the additional twist that the necessary constraint rules can be derived from constraint-based well-formedness rules widely used in the specification of modelling languages ([12]).

IV. Constraint-Based Refactoring

Due to space limitations, we only give a very brief recap of constraint-based refactoring; for more detailed accounts on it, we refer the reader to [10, 11, 13, 14, 18].

In constraint-based refactoring, an artefact to be refactored is represented as a *constraint satisfaction problem* (CSP). The *constraint variables* of this CSP represent properties of the artefact (such as names, types, etc.) that may be changed by the refactoring; the *constraints* represent the rules of the artefact's language's static semantics as applied to these properties, limiting the possible changes to those preserving the artefact's well-formedness and behaviour.

Constraints are generated from an artefact to be refactored by the application of so-called *constraint rules*. For example, the constraint rule

$$\frac{\text{binds(r, d)}}{\text{r.name} = \text{d.name}}$$

expresses that in the artefact to be refactored, for every pair of a reference r and declared entity d such that r binds to d, the name of r and the name of d must be equal. Should a refactoring require the renaming of either r or d, the constraint demands that the other must be renamed with it.

To be able to *specify and implement* constraint-based refactorings declaratively, we developed the refactoring constraint language and framework Refacola [11]. Using Refacola, the following suffices to implement a constraint-based refactoring tool for a target language:

1. a language definition, specifying the elements of the target language, and how these elements are related to constitute the meaning of an artefact (represented by queries, or facts);
2. an implementation of the queries declared in the language definition, which is needed to extract the relationships between elements of the artefact to be refactored;
3. a definition of the constraint rules required to adequately reflect the (static) semantics of the target language; and
4. a definition of the refactorings.

Fig. 1 shows a sample refactoring tool specification.

```
1 language UML
2 kinds
3   Class <: ENTITY { name }
4   Feature <: ENTITY { name, type, owner }
5   Attribute <: Feature
6   Operation <: Feature
7 properties
8   name: Identifier
9   type: ClassDomain
10  owner: ClassDomain
11 domains
12  ClassDomain = [ Class ]
13 queries
14  superclass(c: Class, super: Class)
15 rules
16  uniqueFeatureNames
17  for all
18   F1: Feature   F2: Feature
19   if
20    F1!=F2
21   then
22    superclass*(F1.owner, F2.owner) -> F1.name != F2.name
23 refactoring renameFeature
24   forced name of Feature
25   allowed name of Feature
26 refactoring pullupFeature
27   forced owner of Feature
28   allowed name of Feature, owner of Feature
```

Fig. 1. Refacola definitions of the Rename Feature and Pull Up Feature refactorings for UML class diagrams (excerpt).

V. Model/Code Co-Refactoring

With constraint-based refactorings for the participating modelling and programming languages given, the problem of model/code co-refactoring reduces to connecting refactorings performed on either side so that changes performed by a refactoring on one side propagate to corresponding changes on the other side, where they need to be handled as refactorings in their own right.

A. Cross-Language Constraints

Since constraints propagate changes, they lend themselves to propagating a change in a model to a change in code and vice versa. All that is needed for this is a correspondence of (properties of) model elements to (properties of) code elements. These correspondences are implicitly provided by the code generation process, in which a model element is mapped to a code element. For instance, when a Java field f is generated from a UML attribute a, the correspondence of names is expressed as the constraint

$$\text{f.name} = \text{a.name}$$

If, in addition to the field, accessor methods g (for getter) and s (for setter) are generated, name correspondence is established by the constraints

$$\text{g.name} = \text{"get"} + \text{a.name}$$
$$\text{s.name} = \text{"set"} + \text{a.name}$$

B. Generating Cross-Language Constraints from Transformation Traces

While the constraints required to refactor an artefact in either a modelling or a programming language are generated using rules specific to that language, the constraints required to propagate changes from a model to its program and vice versa must be generated using rules specific to the mapping between the two. This can be done using a trace model.

```
29 language Generation
30 imports Java, UML
31 queries
32   attribute2field(a : UML.Attribute, f : Java.Field)
33   attribute2getter(a : UML.Attribute, g : Java.Method)
34   attribute2setter(a : UML.Attribute, s : Java.Method)
35 rules
36   attributeToFieldNames
37     for all
38       a    : UML.Attribute
39       f    : Java.Field
40       g, s : Java.Method
41     if
42       attribute2field(a, f) and attribute2getter(a, g)
43         and attribute2setter(a, s)
44     then
45       f.name = a.name
46       f.accessibility = private
47       g.name = "get" + a.name
48       g.accessibility = public
49       s.name = "set" + a.name
50       s.accessibility = public
51   …
```

Fig. 2. Refacola specification for the cross-language constraints to be generated from a trace model

A *trace model* is a model capturing mappings of the elements of one artefact, the source of transformation, to elements of another artefact, the target of transformation [19]. For instance, when a field f and its accessors g and s are generated from an attribute a, the trace model contains the facts attribute2field(a, f), attribute2getter(a, g), and attribute2setter(a, s). This can be exploited to generate cross-language constraints using constraint rules of the above kind in exactly the same manner as refactoring constraints are generated. An example of such a constraint rule, along with the declarations of the queries that it uses, is given in Fig. 2.

VI. Model-Driven Development of Co-Refactorings

While our main target so far has been model/code co-refactoring, one might maintain that the distinction between model and code is somewhat arbitrary. And indeed, our approach is equally applicable to code/model co-refactoring, and immediately generalizes to arbitrary pairs of languages whose artefacts are linked by generation, compilation, or some other transformation. In particular, it generalizes to the co-refactoring of domain-specific languages (DSLs) that are compiled to a target language in which the surrounding code is written (and in which the DSL is embedded).

A. Generation of Language Definitions

The language definition part of REFACOLA is a meta(modelling) language for the languages in which is to be refactored. Indeed, every REFACOLA language specification must

- list the *kinds* (*metaclasses* in metamodelling jargon) of the elements of which an artefact is composed (classes, attributes, etc.; cf. Fig. 1),
- assign a set of *properties* (*features* in metamodelling) to each kind (e.g., name, type, etc.); and
- specify the relationships between the instances of each kind and that of other kinds (the *queries*) [11].

In fact, the metalanguage elements used by REFACOLA (kind, property, domain, query) directly correspond to the elements of Ecore (EClass, EAttribute, EType, and EReference). If the DSL is specified using the EMF, this allows the generation of a REFACOLA language specification directly from a DSL specification using model-to-model transformation. We have exploited this in our implementation (cf. Section VII).

B. Generic Querying

Querying against a Java program as required by constraint-based refactoring typically requires a search of the abstract syntax tree (AST), usually involving some amount of program analysis. The situation is very different for DSLs, if the DSLs are specified using a metamodelling framework such as EMF: In that case, the artefact is stored as an instance of the metamodel, in which all relationships are explicit (in Ecore expressed as instances of EReference). Hence, we not only have that the signature of the queries needed for constraint-based refactoring can be generated from the metamodel (cf. above), but also a fully generic query engine can be implemented that evaluates the queries against the metamodel (Ecore) representation of the artefacts to be refactored. Again, we have exploited this in our implementation.

Querying the transformation trace as required for the generation of the cross-language constraints is similarly generic, provided that the code generation produces such traces. If a transformation language such as QVT/R [9] or Mitra [18] (for model-to-model transformation) or Acceleo [1] (for model-to-code transformation) is used, generation of the trace is free.

VII. Implementation

A. Extensions of REFACOLA

We have implemented model/code co-refactoring as described in the previous sections as an extension to REFACOLA. For this, we had to extend the REFACOLA language so that it allows language specifications of different languages (here: UML and Java) to be imported to a module that defines the cross-language queries and constraint rules needed for co-refactoring (Fig. 2).

B. String Constraints

String constraints such as the ones presented in Section V.A are a known problem for constraint solvers (see, e.g., [6]). Since REFACOLA maps all constraint variable domains, including the domain of identifiers, to integers [11], we extended the Identifier domain (whose elements are drawn from the artefact to be refactored) with all relevant string concatenations (forming, e.g., the identifiers of accessor methods), and created a fact base with ternary concat facts stating the concatenations. For instance, the constraint rule for tying the name of a getter to the name of its encapsulated field is written as

```
for all
  a : UML.Attribute
  f : Java.Field  g : Java.Method
  p : String.Get  c : String.Concatenation
if
  attribute2field(a, f) and attribute2getter(a, g)
then
  concat(p, f, c) implies g.name = c.name
```

where p and c are elements of a String language having the kinds Get (a singleton) and Concatenation, whose only property is name.

TABLE I. PROJECTS USED IN THE EVALUATION

PROJECT (Incl. Version)	SOURCE *Model* Classes	Attributes	Operations	*Code* Classes	Fields	Methods
org.gadberry.jexel 1.0.0b 13	45	3	82	117	24	181
org.apache.commons.cli 1.2	18	28	75	112	73	256
Jester 1.37b	78	17	238	240	141	570
JUnit 3.8.2	43	30	239	350	178	832
org.apache.commons.io 1.4	71	21	194	211	178	780
org.jaxen 1.1.5	162	39	807	362	341	1461
org.sinaxe 1.0-b1	158	99	779	571	689	2014
org.snmp4j 1.11.3	161	219	860	399	939	1951
org.dom4j 1.6.1	151	154	1506	449	522	3032
org.htmlparser 1.6	149	268	1195	498	582	2119
total	*1036*	*878*	*5975*	*3309*	*3667*	*13196*

C. Use of Model-Driven Engineering

As suggested in Section VI, significant parts of the specifications necessary for implementing co-refactorings can be generated from the specifications of the involved languages. For this purpose, we have written a Mitra [18] model-to-model transformation that converts any language specification based on Ecore into a REFACOLA language specification. This transformation allowed us to automatically generate REFACOLA language specifications for UML (as implemented in Ecore by MDT-UML2 [7]) and for Ecore itself (which is specified using Ecore). For Java, we used our own, manually written REFACOLA specification and query engine (having 144 kinds, 24 properties, 13 domains, 36 queries, and 132 rules).

We exploited MDE even further when we built refactoring tools for both the Ecore and the REFACOLA language. Because the REFACOLA language is an Xtext-based DSL that has an Ecore-based language model, and because we have a transformation from arbitrary Ecore language specifications (and hence the specifications of Ecore and REFACOLA) to REFACOLA (cf. above), implementing the RENAME and PULL UP FEATURE refactorings for both Ecore and REFACOLA required only the writing of the constraint rules. In addition, with our model-to-model transformation and the trace produced by Mitra, implementation of corresponding Ecore/REFACOLA co-refactorings required only the definition of the necessary cross-language constraint rules. We used this co-refactoring to refactor the Ecore-specification of REFACOLA (using our generated refactoring tools) so that the REFACOLA specification of REFACOLA was refactored with it.

VIII. EVALUATION

We limit evaluation in this section to that of model/code co-refactoring (specifically: UML/Java) as described in Section V; evaluation of the model-driven engineering of refactoring tools as described in Section VI has been evaluated — in the form of two case studies — as described in Section VII.C.

A. Experimental Setup

For our evaluation, we need a set of sample models together with the code generated from them, extended or used by additional code that is not generated, but likely affected by model refactoring. Due to the scarcity of complete MDE projects in the public domain[2], we adopted a reverse engineering approach and generated models from ten open source Java projects. Table I shows the subject projects along with some measures of their size.

We selected three refactorings for our evaluation: RENAME, PULL UP FEATURE, and GENERALIZE DECLARED TYPE. RENAME is by far the most frequently used refactoring in programming [8] and we assume that this holds for modelling also. PULL UP FEATURE is a variant of the more general MOVE FEATURE refactorings that strikes a good balance between opportunities for application and success. GENERALIZE DECLARED TYPE is a special case of a typical model refactoring in which one end of an association (a reference) is redirected from one target class to a superclass (a drag and drop edit facility of many graphical modelling tools).

We applied each of the three above refactorings to every possible location in each model. More specifically, we applied

- RENAME to every class, attribute, and operation;
- PULL UP FEATURE to every attribute and operation of a class that had a superclass that was not from a library, using every such superclass as the target of the pull-up; and
- GENERALIZE DECLARED TYPE to every attribute and operation (in case of the latter generalizing the return type).

With every application, we counted

- whether it was successful, i.e., whether the intended refactoring could actually be performed (tantamount to solvability of the generated constraint system);
- how many constraints were generated on the model and on the code side;
- how long it took to generate and solve the constraints;
- how much of that time was used by querying; and
- if the constraint were solvable: how many changes were implied on the model and on the code side.

B. Results

The results, listed separately per subject program and per refactoring, are shown in Table II. As can be seen, success rates, constraints generated, time taken, and changes induced, vary widely both per program and per refactoring (so that we did not further aggregate them). However, we can see that across all projects and kinds of refactorings, refactoring applications are rejected, and when they are not (the successful applications), on average they require changes in both model and code. This provides evidence that problems of the kind motivating our work (in Section II) actually exist in practice, and that tool support for model/code co-refactoring that addresses these problems is indeed needed.

[2] For none of the open source MDE projects found we could reconstruct the tool chain necessary to parse the models *and* reproduce code generation (necessary to obtain the transformation trace).

TABLE II. Results of Application per Project and Per Refactoring

Project / Refactoring	Applications	Successful	Constraints				Time [msec][§]				Changes (success only)			
			model		*code*		*total*		*querying*		*model*		*code*	
			∅[†]	σ[$]	∅	σ	∅	σ	∅	σ	∅	σ	∅	σ
org.gadberry.jexel 1.0.0b13	319	118	174	556	28	63	248	534	177	423	1.8	2.5	6.5	10
org.apache.commons.cli 1.2	171	130	25	117	8	37	151	325	110	282	1.6	1.9	5.4	5
Jester1.37b	337	264	10	129	7	90	270	1054	200	936	1.8	1.2	4.5	4
JUnit 3.8.2	468	284	81	518	24	120	769	2634	493	1618	1.7	1.4	5.1	8.4
org.apache.commons.io 1.4	444	124	3	21	8	47	229	407	149	257	1.2	0.6	6.1	10.2
org.jaxen 1.1.5	1406	801	97	1832	15	214	1367	8284	989	6248	4.6	6	9.1	12.2
org.sinaxe 1.0-b1	1298	841	63	912	29	317	4010	23327	2087	7868	4.2	6.7	9.5	11.3
org.snmp4j 1.11.3	2031	1055	634	3974	137	658	10382	43689	6212	27234	4.1	5.1	14.3	22.6
org.dom4j 1.6.1	4728	1442	149	2583	51	526	4061	24096	2556	12578	2.5	2.6	12.7	24.1
org.htmlparser 1.6	2813	1488	405	5758	39	494	4839	33260	3652	26145	3.7	7.1	9.2	13.8
Generalize Declared Type	741	158	4528	14165	480	1358	32883	95473	23008	68904	3.6	4.2	9.3	11.5
Pull Up Feature	4787	215	20	303	76	573	3496	25546	1501	9390	1	0.2	1.1	0.5
Rename	8487	6174	0	0	0	0	2425	4044	1800	3387	3.5	5.4	10.6	17.8

[§] all times measured on Amazon EC2 Windows Server instances; evaluations on a contemporary laptop ran ca. 2.5 times faster [†] average [$] standard deviation

Table II also tells us that the number of constraints generated for each refactoring by the REFACOLA refactoring engine is remarkably low, even for the larger projects (it is consistently 0 for RENAME!). This is due to the special constraint generation algorithms described in [11, 13], without which numbers would be in the millions. However, these algorithms lead to rather long times required for constraint generation and solving: more than a minute in some cases may be deemed unacceptable. And yet, the long times are relativized by the long querying times (which take up 64% of the time on average across all applications; note that our generic querying is not optimized in any way), and by the fact that all evaluations were virtualized on Amazon EC2 instances (which, according to our own tests, take about 2.5 times longer than the laptops we are using). Also, our (somewhat brute force) implementation of string constraints (cf. Section VII.B) takes its toll.

IX. Conclusion

Where models are the primary artefacts of software development, model/code co-refactoring is necessarily an issue. Starting with an observation of the essential challenges of model/code co-refactoring, we have extended constraint-based refactorings with bridge constraints that capture the correspondences between model elements and the code elements that are generated from them, propagating changes on either side to the other. Our approach is fully declarative, and significant parts of the necessary declarations can be generated from the (metamodel) specifications of the modelling and programming languages involved in the co-refactoring. While genericity and declarativeness have their price (mostly in terms of performance), at the very least we have shown that model/code co-refactoring can be trusted to a tool, and that the complexity of the tool can be mastered by reusing existing refactoring specifications, reducing the required effort mostly to the design of cross-language constraint rules.

Acknowledgements

This work has been made possible by DFG grant STE 906/4-2.

References

[1] Acceleo Model-to-Text Transformation (www.eclipse.org/acceleo).

[2] L Angyal, L Lengyel, H Charaf "Novel techniques for model-code synchronization" *ECEASST 8: ERCIM Symposium on Software Evolution* (2007).

[3] EMF Metamodel (Ecore) www.eclipse.org/modeling/emf.

[4] J Estublier, T Leveque, G Vega "Evolution control in MDE projects: Controlling model and code co-evolution" in: *Proc. of FSE* (2010) 431–438.

[5] M Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999). ,

[6] A Kiezun, V Ganesh, S Artzi, PJ Guo, P Hooimeijer, MD Ernst "HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars" *ACM Trans. Softw. Eng. Methodol.* 21(4): 25 (2012).

[7] MDT/UML2 (www.eclipse.org/uml2/)

[8] ER Murphy-Hill, C Parnin, AP Black "How we refactor, and how we know it" *IEEE Trans. Software Eng.* 38:1 (2012) 5–18.

[9] OMG *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification* Version 1.1 (http://www.omg.org/spec/QVT/1.1).

[10] F Steimann, A Thies "From public to private to absent: Refactoring Java programs under constrained accessibility" in: *Proc. of ECOOP* (2009) 419–443.

[11] F Steimann, C Kollee, J von Pilgrim "A refactoring constraint language and its application to Eiffel" in: *Proc. of ECOOP* (2011) 255–280.

[12] F Steimann "Contraint-based model refactoring" in: *Proc. of MODELS* (2011) 440-454.

[13] F Steimann, J von Pilgrim "Constraint-based refactoring with foresight" in: *Proc. of ECOOP* (2012) 535–559.

[14] F Steimann, J von Pilgrim "Refactorings without names" in: *Proc. of ASE* (2012) 290–293.

[15] G Sunyé, D Pollet, Y Le Traon, JM Jézéquel "Refactoring UML Models" in: *Proc. of UML* (2001) 134–148.

[16] F Tip, A Kiezun, D Bäumer "Refactoring for generalization using type constraints" in: *Proc. of OOPSLA* (2003) 13–26.

[17] F Tip, RM Fuhrer, A Kiezun, MD Ernst, I Balaban, B De Sutter "Refactoring using type constraints" *ACM Trans. Program. Lang. Syst.* 33(3):9 (2011).

[18] J von Pilgrim *Computerunterstützte Modelltransformationen* Doctoral Dissertation (Fernuniversität in Hagen, 2011).

[19] S Winkler, J von Pilgrim "A survey of traceability in requirements engineering and model-driven development" *Software and System Modeling* 9:4 (2010) 529–565.