Towards Refactoring of Rule-Based, In-Place Model Transformation Systems

Gabriele Taentzer, Thorsten Arendt
Philipps-Universität Marburg
Germany
{taentzer, arendt}@informatik.uni-marburg.de

Claudia Ermel
Technische Universität Berlin
Germany
claudia.ermel@tu-berlin.de

Reiko Heckel
University of Leicester
UK
reiko@mcs.le.ac.uk

ABSTRACT

The more model transformations are applied in various application domains, the more questions about their quality arise. In this paper, we present a first approach towards improving the quality of endogenous in-place model transformation systems. This kind of model transformations is typically rule-based and well suited to perform model simulations and optimizations. After discussing suitable quality aims for this kind of model transformation systems and how they can be detected by smells, a first selection of refactorings is presented showing a variety of potential improvements of model transformation systems. Each refactoring is presented in a systematic way including an explanation how the quality is improved, a description of its pre- and post-conditions, a possible refactoring strategy, and an example. All discussed refactorings are implemented in Henshin, a model transformation engine based on graph transformation concepts, using Henshin in combination with the Eclipse plug-in EMF Refactor, a refactoring plug-in for defining and applying refactorings of EMF models.

1. INTRODUCTION

Model transformations have been applied to solve various tasks in model-driven engineering (MDE) such as model refactoring and optimizations, translation into other modeling languages, simulation and analysis, model migration and code generation [17].

As model-driven technologies are becoming more mature, it is worthwhile to make explicit expert knowledge on how to create and maintain model transformation systems. In this paper we address the refactoring of model transformations. Originally, refactoring means to improve a program's structure without changing its behavior [11]. Meanwhile, this technique has also been used to improve other kinds of software artifacts, such as models.

In a previous paper [8], we considered how changes in domain models can imply changes in rule-based model transformations. In this paper, we generalize these ideas by providing a first collection of refactorings for in-place model transformation systems, motivated by certain quality aspects. While model translations are typically *out-place*, i.e., constructing new result models, endogenous model transformations (sticking to one language) may also be *in-place*, i.e., modifying the input model directly [6]. Model simulation and refactoring as well as other kinds of model modifications such as further model optimizations and advanced editing operations are typically realized by in-place transformations. Note that we consider refactorings of in-place model transformations in this paper, since our refactorings do not refer to either source or target model elements only. Refactorings of out-place model-to-model transformations are presented in detail in [18]. Nevertheless, we could also apply our techniques and tool to out-place (model-to-model) transformations, which can be emulated by considering an integrated domain model constructed from the source and target domain, and defining a rule set where only target domain model elements are generated [10].

Since specifications of model transformations are software models, we adapt well-known quality assurance techniques for models [4], based on metrics and smells to determine quality aspects. Refactorings are adapted to domain- and project-specific needs, including the specification of new ones. Such a project-specific quality assurance process is applied to models until their quality is sufficiently improved. Adapting this approach to model transformation systems, we identify quality aspects for model transformation systems and introduce suitable smells (potential indicators of low quality) and refactorings that make existing knowledge explicit about how to write model transformation systems.

Our selection of refactorings is guided by mainly two aspects: First of all, we concentrate on what we consider the core concepts of rule-based, in-place transformation systems. Considered concepts are meta-models with their usual object-oriented features, and rules with left- and right-hand sides as well as positive and negative application conditions. Furthermore, refactorings have been selected according to certain quality aspects. We concentrate on *conciseness*, *comprehensibility* and *changeability* of model transformation systems. Quality improvements are indicated by a variety of smells being based on metrics and patterns.

The main contribution of this paper is a first collection of useful refactorings for rule-based, in-place model transformation systems, described in a systematic way. To integrate these refactorings into a systematic quality assurance process, we discuss quality aspects for model transformation systems and define a first collection of smells based on metrics and patterns. Each refactoring description is presented

by means of a short description, an explanation in which ways the quality is improved (affected smells), a description of its pre- and post-conditions, a possible refactoring strategy, and an example. Furthermore, we present ideas on how to implement presented refactorings on the basis of HENSHIN [2], a model transformation engine for the Eclipse Modeling Framework (EMF) [15] based on graph transformation concepts.

Structure of the paper: In Section 2, the core concepts of rule-based, in-place model transformation systems are discussed. Valuable quality aims for this kind of model transformation systems and smells affecting these quality aspects are discussed in Section 3. Section 4 presents our refactoring collection. In Section 5 we present ideas for the implementation of Henshin refactorings using HENSHIN again. Finally, we conclude the paper in Section 6.

2. CORE TRANSFORMATION CONCEPTS

The core concepts of rule-based, in-place model transformation approaches form the basis for our catalogs of smells and refactorings. Of course, taking further concepts into account, the corresponding transformation language is widened and the catalogs shall be extended accordingly. Transformation languages offering (most of) these core concepts are, e.g., Henshin [2], ViaTra [16], Groove [12], and ATL (in-place) [5].

An instance model consists of a set of *objects* having *attributes* and *references*. While attributes are typed over data types, references are typed over classes. All instance models have to conform to a domain or type model, also called meta-model, supporting *class inheritance*, including *abstract classes* (without instances) and *containment relations*. As example, consider the domain model for phones in the upper leftmost screen shot in Figure 1. Due to space limitations, we do not discuss multiplicities and further constraints here.

Transformation rules specify local changes on instance models. Usually, a *rule r* contains two model patterns, called left-hand side (LHS) specifying the pre-condition and right-hand side (RHS) formulating the post-condition of the rule.Either the differences between LHS and RHS show us the modifications induced by the rule (as in Henshin and ViaTra) or all modifications are defined in the RHS only (as in ATL). Alternatively, one pattern may be given being an integration of both rule sides where elements and references to be deleted or created are annotated accordingly. In addition, checks and computations of attribute values can be specified by expression languages such as JavaScript and OCL. A rule is applicable to some model if the left-hand side pattern occurs in the model[1] or the guard pattern is satisfied, including the satisfaction of all attribute value checks. In that case, all specified rule actions are performed[2]. Rule elements may be typed over abstract classes, however, when applied, each rule element has to be mapped to some model element concretely typed. Rule elements specifying object creation have to be typed concretely already in the rule. Furthermore, variables for attribute values may be defined in the scope of a rule to be used for checks and computations. When a rule is applied, its variables are bound to concrete data type values.

The application of a rule may be further restricted by conditions being any kind of logical expression over the existence of model patterns. In the following, we restrict our considerations to the most simple ones being used by graph transformation-based approaches, i.e., *negative application conditions* (NACs) and *positive application conditions* (PACs) which forbid resp. require the existence of certain model patterns in instance models the rule is applied to.

Example. Figure 1 shows an example using Henshin: A simple domain model for phone systems is shown together with rule *liftFixed* for lifting a fixed phone. The only effect of this rule is to unset attribute *isIdle*. This rule is applied to a simple instance model shown underneath using EMF-Compare [9]. Note that the transformed instance model is shown on the left, while the original one is on the right.

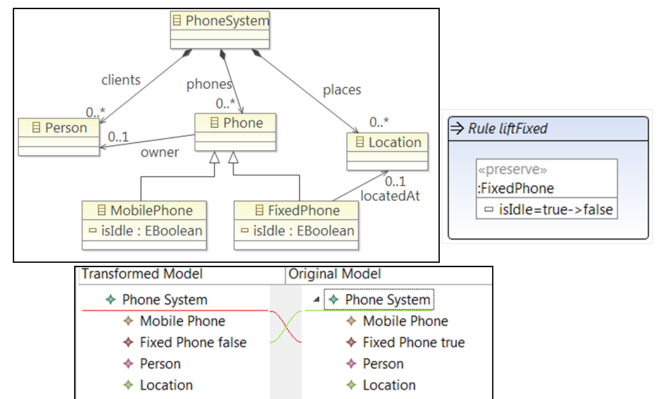

Figure 1: Domain model, rule, and a transformation step in Henshin

3. QUALITY ASPECTS AND SMELLS FOR MODEL TRANSFORMATION SYSTEMS

In this section, we motivate quality aims for model transformation systems as explained above, and give structured descriptions of some smells affecting these quality aspects.

3.1 Quality aspects

As for other software artifacts, the *correctness* of a model transformation system is defined w.r.t. the transformation language used and its interpretation in terms of the domain. While *language correctness* is considered syntactical, the interpretation forms the *model semantics*. Refactorings are supposed to preserve the model semantics.

Conciseness is concerned with the compactness of models which should present systems on the right abstraction level. It is open how to measure conciseness effectively. We can consider the size of transformation models, i.e., the size of domain models and the numbers of rules and rule elements. Sticking to a level of abstraction, we can say that the smaller these numbers are, the more concise is the model. A discussion on model transformation metrics can be found in [1].

[1] We restrict to injective matching of the LHS.

[2] Formally, we follow the DPO graph transformation approach for rule application [7].

A model transformation system is *changeable*, if it can be evolved rapidly and continuously. Conciseness and moreover, low redundancy and low coupling of modules, seem to be necessary prerequisites for changeability.

A model transformation system is *comprehensible* if it is understandable by the intended users. *Comprehensibility* is increased if a system is simple, concise, and structured enough to grasp its design. Moreover, comprehensibility is also influenced by the quality of the used concrete syntax (textual or graphical layouts), however, we do not consider this quality aspect throughout this paper.

3.2 Selected smells

In the following, we present a small set of selected smells for rule-based, in-place model transformation systems. Smells report on suspicious system parts which should be inspected closer. Since we are mainly interested in the conciseness, comprehensibility, and changeability of model transformation systems, we investigate size and redundancy issues. Each smell is described in a structured way including affected quality aspects and refactorings that can eliminate them (the refactorings are described in detail in Section 4).

Smell "Large Rule": A rule specifies a model pattern and replaces it. It should handle a single aspect of the behavior. A large rule seems to care about too many different concerns.

Detection: This smell can be easily detected by counting the number of elements in a given rule. This smell depends very much on the modeling purpose: First, it has to be decided if objects, relations, pre-conditions, or actions are counted. Second, the threshold value has to be determined by experimental investigations.

Affected quality aspects: Large rules do not represent a good modular design and can contain redundant information. Conciseness and comprehensibility might be affected.

Usable refactorings: Extract Pre-condition, Loop Edges to Boolean Attributes;

Smell "Redundant Attributes and References": Several model element types have equivalent attributes and references.

Detection: This smell can be detected by comparing the number of all attributes and references and the number of equivalent attributes and references.

Affected quality aspects: Redundant information blows up the meta-model and potentially also the rule set. It affects the conciseness, comprehensibility, and changeability of model transformation systems.

Usable refactorings: Pull Up Attribute, Pull Up Reference;

Smell "Redundant Rules": Several rules with equal pattern structures may differ in model element and attribute types used only.

Detection: This smell can be detected by comparing the number of all rule pairs differing in types only.

Affected quality aspects: Redundant information blows up the meta-model and the rule set. It affects the conciseness, comprehensibility, and changeability of model transformation systems.

Usable refactorings: Pull Up Attribute, Pull Up Reference, Abstract Rule;

Smell "Unused Object Type": There are object types that are not used in rules at all. Here, the purpose of the transformation rule set has to be considered when interpreting this smell (e.g., transformation of the entire model vs. local transformation).

Detection: This smell can be detected by counting the rules using a specific object type.

Affected quality aspects: Unused object types may affect the correctness, the completeness and the conciseness of of transformation systems, dependent on the reason for this smell. Wrong types may be used, rules may be missing, or types may not be needed.

Usable refactorings: Eliminate Object Type, Change Object Type;

Smell "Delete and Create the Same Object": There are rules with objects being first deleted and then created again with the same attribute values but different contexts, or the same contexts but different attribute values.

Detection: This smell can be detected by applying some clone detection to find corresponding patterns in rules.

Affected quality aspects: If objects are deleted and immediately created again keeping their attribute values or their contexts, rules are not as concise and comprehensible as possible and can be improved.

Usable refactorings: Move vs. Delete / Create;

Smell "Rules With Common Subrule": The model transformation system has several rules containing the same subrule.

Detection: This smell has to apply some clone detection to find common subpatterns in rule parts.

Affected quality aspects: If rules have common subrules, they contain redundant information that may affect quality aspects such as conciseness, changeability, and comprehensibility.

Usable refactorings: Unify Rules with Same Actions;

Further smells are the well-known object-oriented smells that may be checked on the meta-model having also effects on the rule set in general.

4. SELECTED REFACTORINGS

In this section, we present a collection of refactorings for rule-based, in-place model transformation systems, each described in a systematic way. This collection mirrors our experiences in the application of model transformation to various purposes. It shows a range of refactorings serving

several quality aims. For example, refactoring ”Pull Up Attribute” reduces the amount of redundancy w.r.t. attribute definitions and potentially also reduces the number of rules, while ”Extract Pre-condition” reduces the number of rule elements and thus improves the conciseness. Each refactoring is systematically described including an example and change of identified smells before and after a refactoring, and an argumentation how semantics is preserved. For model transformation systems, semantics preservation may refer to the preservation of model transformation sequences, the preservation of transformed models, or the preservation of the amount of information in models.

Note that we do not present a refactoring which is probably most useful, i.e., the renaming of transformation systems, rules, types, etc., since it is obvious. Furthermore, the well-known refactorings of object-oriented models such as Extract Superclass, Pull Up Attribute, Remove Middle Man, etc. are basically applicable to domain models to improve them. Changes in domain models can imply changes in rules [8]. It may happen that rules differing in types only can be merged by using a superclass as type. Furthermore, most of the refactorings presented below come with an inverse, taking back the original refactoring effect. E.g. the inverse refactoring of “Pull Up Attribute” is ”Push Down Attribute” which might be useful to prepare a variation of attribute definitions in subtypes. The inverse of “Extract Pre-Condition”, called “Inline Pre-Condition”, may be helpful for rule modifications. In this paper, inverse refactorings are not presented in detail, due to space limitations.

4.1 Refactoring “Merge Rules Differing in Types Only”

If there are rules which differ in object types only and these types are subclasses of the same superclass, they can be merged to one rule. This refactoring is often combined with a “Pull Up Attribute” refactoring of the domain model.

Input parameter: Names of the rules to be merged.

Example: Phones are refined to fixed and mobile phones. Both subtypes are attributed by a Boolean attribute *isIdle*. Two rules describe the lifting of fixed resp. mobile phones (see Figure 2 with domain model in Figure 1). A refactoring “Pull Up Attribute” is performed on the domain model first to pull attribute *isIdle* up to class *Phone* (if class *Phone* does not have the *isIdle* attribute already). Figure 3 shows the desired domain model and contains a lift rule for phones in general being abstracted from the two original lift rules. This is possible, since the rules in Figure 2 differ in types only and thus, can be merged to the rule in Figure 3.

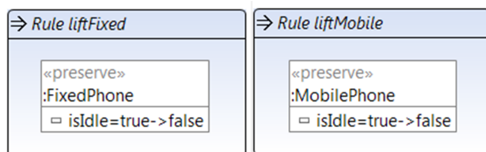

Figure 2: Before refactoring “Merge Rules Differing in Types Only”

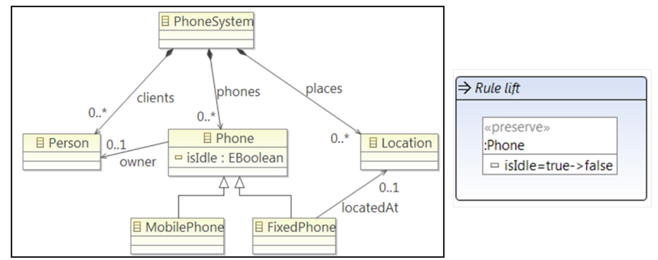

Figure 3: After refactorings “Pull Up Attribute” and “Merge Rules Differing in Types Only”

Pre-condition: Indicated rules differ in one object type only. The set of varying object types found contains all subclasses of a common superclass.

Strategy:

1. Identify all varying object types being classes with a common superclass.
2. Construct a new rule by taking one original rule and replacing identified subclasses by identified superclass. Rename the modified rule, if necessary.
3. Delete all remaining original rules.

Post-condition: All original rules are replaced by one new rule using the identified superclass as object type.

Affected smells: Redundant rules

Quality improvement: The number of rules becomes smaller. The model becomes more concise.

Semantics: The semantics is preserved, since the same transformation sequences are induced.

4.2 Refactoring “Extract Pre-condition”

This refactoring makes pre-conditions explicit by extracting preserved parts as positive application conditions.

Input parameter: name of the rule

Example: A new fixed phone is installed. The rule mainly consists of context, i.e., preserved model elements that are not transformed. We extract the context that is not needed for inserting new edges into a positive application condition to make it more explicit (see Figure 4). Note that this reduces the size of the internal rule representation, though this effect is not visible in our compact notation.

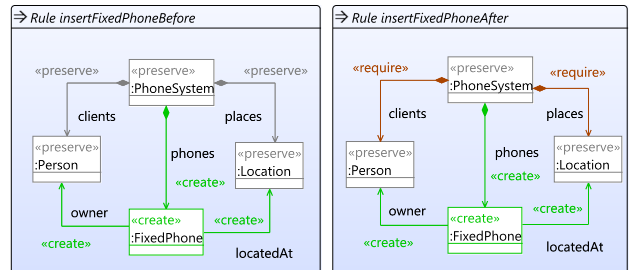

Figure 4: Before and after refactoring “Extract Pre-Condition”

Pre-condition: none

Strategy:

1. Determine the preserved part of the input rule.
2. Create a new PAC and put those preserved objects into it that are not needed as targets for newly created references.
3. Reduce the rule's preserved part to the boundary objects needed for creating new references.

Post-condition: The preserved part of the rule is minimal.

Affected smells: Large Rule, Implicit Pre-condition

Quality improvement: The rule is better comprehensible, since the pre-condition is expressed more explicitly.

Semantics: The semantics is preserved, since the same transformation sequences are induced.

4.3 Refactoring "Move Vs. Delete/ Create"

Rule elements being deleted and created in the original rule, are moved afterwards.

Input parameter: Name of the rule

Example: Taking up the *Phone* example again, we consider a rule that replaces a fixed phone at one location by another one at another location, i.e. the fixed phone at the original location is deleted and a new one is created at the new location. After the refactoring, the rule specifies the movement of a fixed phone from one location to another one (see Figure 5).

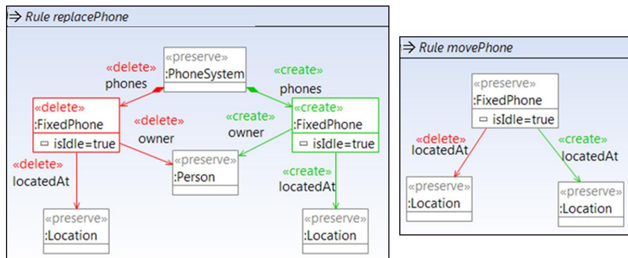

Figure 5: Refactoring of deletion and creation of a fixed phone by its movement

Pre-condition: There are model objects being first deleted and then created again with the same attribute values but different contexts or same contexts but different attribute values.

Strategy:

1. Identify objects and references being deleted and created afterwards. If these elements are attributed, they are either identified if the attribute values of created elements are the same as of deleted ones or if their adjacent references are created in the same way as they existed before.
2. Preserve identified elements instead of deleting and creating them.

Post-condition: The rule does not contain any object that is deleted and created with the same attribute values or the same context.

Affected smells: Delete and Create the Same Object

Quality improvement: The resulting rule is more concise, since unnecessary actions are avoided.

Semantics: The semantics is preserved in the sense that the same models are created, when both rules are applicable; however, the number of transformation effects when applying the refactored rule is reduced. Note that the original rule is not applicabele if the FixedPhone node has more incident edges than specified by the rule (in the DPO approach), whereas the refactored rule is applicable also to FixedPhones with more connections.

4.4 Refactoring "Unify Rules with Same Actions"

Given a set of rules which share a subset of actions. This subset is encapsulated in a new rule to be applied first. The original rules are reduced to their remaining actions each.

Input parameter: Set of rule names

Example: For registering a new phone, it is enough for mobile phones to set the person who will own it. For fixed phones, their location has to be registered in addition. These two cases are specified in rules "registerMobilePhone" and "registerFixedPhone" in Figure 6. However, the owner registration is common to both rules. Thus, we can form a kernel rule for phone registration handling the owner registration only. While this is all what has to be done for mobile phone there is a remainder rule for fixed phones. It specifies the location registration only. We have to make sure that to fixed phones both rules in Figure 7 are applied.

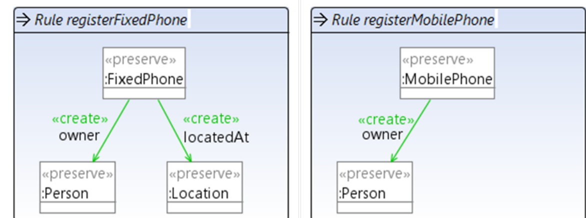

Figure 6: Before refactoring "Unify Rules with Same Actions"

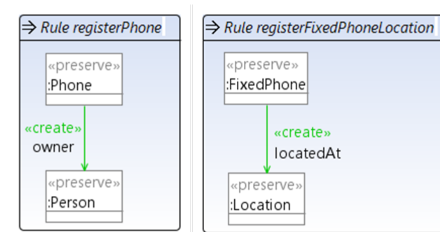

Figure 7: After refactoring "Unify Rules with Same Actions"

Pre-condition: None

Strategy:

1. Identify the set of actions being shared by the set of input rules.

2. Besides the common actions identify also the common preserved model part.
3. Create a new rule, called *kernel rule* containing all identified actions and the identified preserved part. If common actions and preserved parts differ only in all subclasses of a common super class, this common super class is used as object type instead.
4. Reduce each of the original rules, called *remainder rule*, by the identified set of actions. Reduce the preserved part if it is common and not needed for the remaining actions, i.e., if it forms a pre-condition.
5. Make sure that the kernel rule is applied before remainder rules. This can be done e.g. by an additional control structure putting both rules into a sequence.

Post-condition: There is a new rule, the kernel rule, that contains all common actions and the common preserved part. All remainder rules do not contain common actions or pre-conditions anymore. A remainder rule is not applicable without applying the kernel rule beforehand.

Affected smells: Rules With Common Subrule

Quality improvement: The number of elements in the considered rule set is reduced, i.e., its conciseness is increased.

Semantics: Each original rule can be constructed by the composition of the kernel rule and optionally, a remainder rule. There may be more transformation sequences than before, since the resulting transformations allow for more interleaving of rule applications than before.

5. IMPLEMENTATION ISSUES

This section presents the key ideas for implementing model transformation refactorings using the Eclipse Modeling Framework (EMF) [15] as basis to define domain models, and HENSHIN to specify rule-based, in-place transformation systems.

Henshin transformation systems can be refactored in a straightforward way by defining higher-order transformations on the Henshin transformation model as Henshin transformation system again, i.e., refactorings are implemented based on Henshin rules typed over the Henshin transformation model being the domain model in that case.

To specify these refactorings as advanced operations to be integrated in the Henshin editors, we use EMF Refactor [14] supporting the specification and application of refactorings to EMF-based models. Since the Henshin model is an EMF model, EMF Refactor can be used. The specification of refactoring *Extract Pre-condition* can be found at [3].

6. CONCLUSION

Refactorings are a well-established means to make development experiences explicit such that further developers can benefit from these experiences. Therefore, we start with a selection of interesting refactorings for rule-based, in-place model transformation systems. In this paper, we explicitly restrict this approach to core features. However, further features should be considered, primarily rule parameters and control structures for rule applications. In addition, it is certainly worthwhile to define not only metric-based but also pattern-based model transformation smells. Here, novel smells that are specific to transformations have to be considered. Taking also typical smells for out-place model-to-model transformations into account [18], more complex nested and structured refactorings have to be considered. Here, our underlying basis of graph transformation theory may help to analyse complex refactoring rule systems for conflicts and dependencies among rules [13]. It is up to future work, to enable developers to define comprehensive quality assurance processes for model transformation systems that can be adapted to specific needs.

7. REFERENCES

[1] van Amstel, M., van den Brand, M., Nguyen, P.H.: Metrics for Model Transformations. In: Ninth Belgian-Netherlands Software Evolution Workshop (BENEVOL 2010), Lille, France (2010)

[2] Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proc. MODELS. LNCS, vol. 6394, pp. 121–135 (2010)

[3] Arendt, T.: Specification of Refactoring Extract Pre-condition for Henshin Transformation Systems, www.mathematik.uni-marburg.de/~arendt/amt12/

[4] Arendt, T., Kranz, S., Mantz, F., Regnat, N., Taentzer, G.: Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support. In: Software Engineering 2011. LNI, vol. 183, pp. 63–74. GI (2011)

[5] ATL Transformation Language, eclipse.org/atl/

[6] Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)

[7] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science, Springer (2006)

[8] Ehrig, H., Ermel, C., Ehrig, K.: Refactoring of Model Transformations. ECEASST 18 (2009)

[9] EMF Compare, eclipse.org/emf/compare/

[10] Ermel, C., Hermann, F., Gall, J., Binanzer, D.: Visual modeling and analysis of EMF model transformations based on triple graph grammars. ECEASST (2012), To Appear

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

[12] Groove, http://sourceforge.net/projects/groove/

[13] Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and System Modeling 6(3), 269–285 (2007)

[14] EMF Refactor, eclipse.org/modeling/emft/refactor/

[15] Steinberg, D., Budinsky, F., Patenostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd Edition. Addison Wesley (2008)

[16] ViaTra, eclipse.org/gmt/VIATRA2/

[17] Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley (2006)

[18] Wimmer, M., Martínez, S., Jouault, F., Cabot, J.: A catalogue of refactorings for model-to-model transformations. Journal of Object Technology 11(2), 2:1–40 (Aug 2012)