First European Workshop on Composition of Model Transformations - CMT 2006

Proceedings

10 July 2006

Bilbao, Spain

http://ew-cmt.ewi.utwente.nl

Edited by
Anneke Kleppe

Organisation Committee

Anneke Kleppe, University Twente, The Netherlands
Ivan Kurtev, INRIA & University of Nantes, France
Jos Warmer, Ordina, The Netherlands

Programme Committee

Klaas van den Berg, University Twente, The Netherlands
Jeff Gray, University of Alabama at Birmingham, USA
Jan Hendrik Hausmann, S&N AG, Germany
Jochen Küster, IBM Research, Switzerland
Jon Oldevik, SINTEF, Norway
Richard Paige, University of York, UK
Alfonso Pierantonio, Università degli Studi dell'Aquila, Italy
Andy Schürr, Darmstadt University of Technology, Germany
Antonio Vallecillo, University of Malaga, Spain
Daniel Varro, Budapest University of Technology and Economics, Hungary

Contents

Preface

This technical report contains the proceedings of the first European Workshop on Composition of Model Transformations held in Bilbao, Spain on July 10, 2006, as a satellite event to the European Conference on Model Driven Architecture.

Model transformation techniques have been studied for some years now and they have advanced quite a lot. More recently, attention has been given to the subject of transformations composition. One possibility for transformation composition is chaining several model transformations potentially expressed in different languages and executed by different tools. Another possibility is to compose rules from two or more existing transformations usually written in the same transformation language into a new transformation. The latter possibility may follow a composition of existing metamodels.

Because of the nature of model transformations, which is to implement an m-to-n relation between models, a simple composition mechanism like the Unix pipe does not suffice. Another complication of composition of model transformations is that it involves interoperability between various transformation tools.

Composition of transformation rules into a new transformation requires proper modularity constructs and compositional operators within a single transformation language. Most languages provide constructs similar to the well-known constructs in programming languages: inheritance, aggregation, templates, etc. But these constructs have also some limitations. We need a better understanding of the nature of the pieces of transformation functionality that must be modularized, reused, and composed.

The aim of the workshop is to identify the research issues and the existing work in this area. We tried to cover the topic in breadth and therefore invited short papers (position statements) only. We are very happy with the number and quality of the submitted papers, which are presented in this technical report.

July 2006

Anneke Kleppe, Ivan Kurtev, and Jos Warmer

A Framework for Transformation Chain Development Processes

Bert Vanhooff, Dhouha Ayed, and Yolande Berbers

Department of Computer Science, K.U. Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
{bert.vanhooff, dhouha.ayed, yolande.berbers}@cs.kuleuven.be

Abstract. Model Driven Development (MDD) promotes the use of abstract models in software development. A key ingredient of MDD is the application of transformations to these models, which means that part of the development effort is relocated to the transformations. Currently there is almost no available guidance to help designing a suitable, project specific, transformation chain. We propose a framework of four concern layers to organize transformations, which facilitates better separation-of-concerns and offers opportunities for transformation reuse and replacement. We use this framework as a foundation to build a incremental transformation chain design process.

1 Introduction

Model Driven Development (MDD) is an approach to developing software that proposes using machine-readable models at various levels of abstraction as its main artifacts. The key MDD idea is to (semi-)automatically transform highly abstract models into more concrete models from which an implementation can straightforwardly be generated.

However, it is not enough to have a set of interesting development ideas and concepts to create great software; the way we use these is just as important. For example, it is not just because we implement an application using an object oriented language that our software is automatically well structured. The same is true for MDD: it is not because we apply an MDD approach that we get a good system. A good design of the system helps development just as much, if not more, as the development paradigm we use to realize it.

In this paper we argue that it is possible to achieve a better separation of concerns than with the classical PIM/PSM (Platform Independent/Specific Model) distinction (Section 2). In Section 3 we introduce a framework for transformation development processes that can be refined with concrete activities. We wrap up by presenting related work (Section 4), drawing some conclusion and discussing future work (Section 5).

2 Separation of Concerns with Transformations and Abstract Platforms

Many introductions on MDD use the notions of PIM and PSM, which were introduced by OMG's MDA [1]. A PIM is a model of a system that contains no technical details while a PSM is an elaboration of the same system that contains exactly these details. A single-level transformation process between PIM and PSM allows us to capitalize on stable platform independent matters and generate PSMs for a range of different concrete technology platforms.

The use of transformations can provide a more fine grained sense of separation-of-concerns than is implied by the black-and-white PIM/PSM separation. A *transformation chain* specifies how a number of transformations work together, each elaborating on the source model to come to a target model. The following types of concerns could subsequently be addressed in a transformation chain:

Functional concerns influence a model at the highest level of abstraction, namely the application business domain. For transformations this can mean automatic insertion of additional (pluggable) functionality into the basic model and can be accomplished with what is called model composition or weaving. – e.g. merge a basic chat application model with a model for providing file transfer.

Non-functional concerns determine the quality characteristics of a system. We consider them in a technology independent way at this stage. This means that transformations at this level can only apply generic solutions for a non-functional concern, without referring to platform-specific solutions. – e.g. a general distribution model, general persistence specification.

Technical concerns worry about the realization of the above (non-)functional concerns by using technology specific mechanisms (middleware concepts). – e.g. distribution realized with CORBA, persistency with an RDBM.

Implementation concerns go further than middleware-related concerns. Transformations for this category prepare a model for implementation in a certain programming language. – e.g. CORBA in Java, PG/SQL as RDBM.

The above categorization depicts a four-layered transformation approach where each layer can contain a number of concern-specific transformations that facilitate a gradual move from platform independency to platform specificity. The layers introduce constrained boundaries for transformations, forcing a narrower focus for each transformation and therefore taking one step towards easier reuse and replacement.

Each intermediate model can be seen as being specific to a virtual platform, corresponding the a concern layer, but independent to platforms further up the transformation chain. Abstract platforms are introduced in [2] and can serve as boundaries between transformations. A simple example of abstract platform is a user-defined subset of the UML with all semantic variation points fixed.

3 Transformation Chain Development Process

The discussion in the previous section provides a certain mind set to start thinking about transformations. Nevertheless this is often not enough to design a good transformation chain. We need some additional guidance that helps us decide which concrete models, notations and transformations are needed in the same way that classical software development processes, like the Unified Process (UP), offer guidance for classical application development starting from a set of requirements.

We expect a transformation chain design process to produce a model that has the following characteristics:

- Mutually exclusive transformations – Avoid overlap between transformation.
- Clearly separated transformation (concern) areas – A concern can easily be traced to a (group of) transformation(s).
- Loose coupling – Transformations do not rely on each other's implementation properties but only on externally defined pre- and postconditions (reusability).
- Technology independence – The transformation chain model is specified independent of the technology that is used to implement it.

We present the outline of an iterative process framework that can act as a starting point for concrete transformation chain design processes based on the notion of the four-layer concern separation defined in Section 2. We name the iterations *inception* and *elaboration* corresponding to UP terminology. The evolution of a transformation chain model throughout the iterations is shown in Figure 1. Model types are shown as square boxes (labeled Mx, gray boxes are additions and hashed boxes are refinements), transformations are indicated as arrows (labeled Tx), abstract platforms are shown as solid lines and concern layers are shown as dotted lines. This example just illustrates the main concepts and is purely fictional.

Inception: MDD-specific Vision and Concerns

A first iteration should clearly state a fair amount of the requirements that we impose on the transformation chain in order to get a common vision. Three important things have to be identified here:

1. Motivation for applying an MDD approach. This states the general quality requirements for the transformation chain and could be elaborated by using some form of transformation use cases. – e.g. offer a domain specific language (DSL) (top left of Figure 1), automate error-prone coding parts.
2. Concerns that you want to address in transformations instead of explicitly specifying them in the main application model – e.g. Distribution and persistence (both non-functional).
3. Pre-fixed constraints for the project that could influence the transformation chain development. – e.g. the use of CORBA and PG/SQL (the first is a technical concern, the second an implementation concern), using the UML as primary modeling language. Some of the constraints can already be indicated

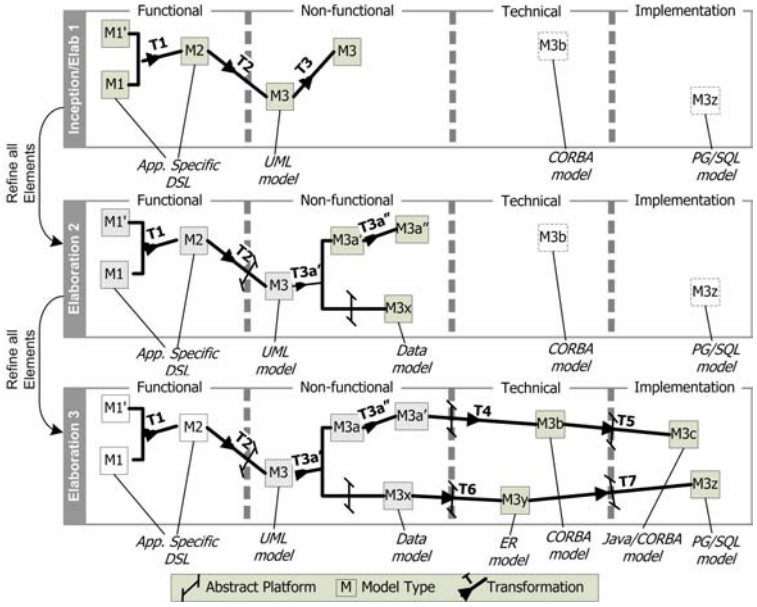

Fig. 1. From top to bottom: incremental development of a transformation chain, showing the four concern layers with transformations, model types and abstract platforms.

on the transformation chain model (M3b and M3z on top of Figure 1 indicate the CORBA and PG/SQL constraints).

Elaboration 1: Conceptual Transformation Chain Model

Since we aim for an incremental top-down approach the first elaboration iteration will primarily address the functional and non-functional layers. The goal here is to identify most of the transformations and model types in this area by describing them informally according to the concerns they address (e.g. persistency transformation, data model, distribution model, etc.) and decompose them according to common subconcerns.

In the top part of Figure 1 we show three transformations: the first one (T1) weaves two models in the application's DSL, T2 translates the DSL models into an equivalent UML representation and T3 represents the persistency transformation concern.

If we identify transformation overlap, e.g. we can have a 'notifier' concept from the persistence concern and an 'event' concept from a logging concern (not shown in the figure), this can be handled by a designated transformation addressing event and notification as separate subconcern. Hence we split T3 in T3a' and T3a" (middle part of Figure 1). This kind of decomposition leads to more focused and potentially more reusable subject-matter specific transformations.

Elaboration 2: Refined Transformation Chain Model
In this iteration, we refine the conceptual model by formalizing the in- and output model types of each transformation and, accordingly, determine the abstract platforms.

We decided for example to let T3a' also output a dedicated data model besides a refined UML model (middle of Figure 1) – another decomposition. If in- and output of a transformation have a different model type, we have to add an platform boundary (between M3 and M3x). Each abstract platform is described by a new or exsiting/modified metamodel.

Finally we can describe the transformations in more detail by using the vocabulary defined in the metamodels/abstract platforms.

Elaboration 3: Fully Specified Transformation Chain model
In the last iteration (in practice there could be more) we execute further transformation decompositions and we consider the technical and implementation layer. Furthermore, the existing chain can be refined at all points in general.

If we consider transformations that operate within the same metamodel (or platform), some more fine-grained decompositions might be possible since each transformation is expressed in detailed metamodel/platform concepts instead of high-level, informal concern concepts (as in elaboration 1). A typical example in the UML domain is the addition of a separate transformation that generates get and set operations because these are UML specific concepts.

To close the gaps to the technical and implementation domain, we introduce for example the Entity-Relationship (ER) model (bottom of Figure 1) between the data model and the pre-determined PG/SQL model. Delaying these layers up till now, improves the separation between them and the higher layers. Furthermore the model types (metamodels) for the lower layers are assumed to be well-known since they correspond with concrete platforms.

4 Related Work

Our work is partly inspired by [3], where the importance of preparation phases (development transformations, metamodels, model notations, etc.) in developing a project-specific MDD infrastructure is emphasized. We addressed one part of such an infrastructure: the transformation chain.

We argued for four conceptual concern layers, each in which many transformations can operate. The Enterprise Fondue method [4] introduces a layering that uses UML profiles to distinguish five abstraction layers: component, concern refinement, technical, platform and implementation. This method is aimed towards distribution and other middleware related concerns. Similarly the RM-ODP (Reference Model for Open Distributed Processing) [5] defines five viewpoints to look at a distributed system: enterprise, information, computational, engineering and technology. Our layering is very similar to the two previous ones but we wish to use it to address a more broad area of concerns. Almeida et al. introduced the notion of abstract platform [2] in particular for separating mid-

dleware specific concerns and discusses the costs and benefits of more levels of abstraction in [6].

We did not discuss technical solutions for decomposing transformations, however some work has been done in this area. In [7] an OCL-based method is offered to formally specify pre- and postconditions for transformations, which is required to make transformations self-contained and easily reusable. In [8] we discussed a UML-based traceability mechanism that provides additional means to split up transformations into smaller, commonly used, pieces.

5 Conclusions and Future Work

An important part of the effort in an MDD-based project lies in the development of an appropriate transformation chain, which in turn eases the construction of the application(s) described in a project.

We introduced a four-layer concern framework for transformation chains that is richer than the classical PIM/PSM separation. To make good use of this framework we presented an initial high-level transformation chain design process that uses the layers to incrementally guide the designer to a complete transformation chain model. The layered approach facilitates a better separation-of-concerns and is a first step towards reusable and replaceable transformation components.

All ideas presented in this paper need to be refined. In our future work we will address concrete approaches for defining the layers formally technical solutions for composing transformations. We will also refine the proposed high-level process with concrete activities that offer better guidance to developers in order to evolve to a full-fledged transformation chain design process.

References

1. Object Management Group: Mda guide version 1.0.1. Misc (2003)
2. Almeida, J.P., Dijkman, R.M., van Sinderen, M., Pires, L.F.: On the notion of abstract platform in mda development. In: EDOC. (2004) 253–263
3. Gavras, A., Belaunde, M., Almeida, L.F.: Towards an mda-based development methodology. In: EWSA. (2004) 230–240
4. Silaghi, R., Fondement, F., Strohmeier, A.: Towards an mda-oriented uml profile for distribution. In: EDOC. (2004) 227–239
5. ISO/IEC and ITU-T: Reference model for open distributed processing. (Misc)
6. Almeida, J.P.A., Pires, L.F., van Sinderen, M.: Costs and benefits of multiple levels of models in mda development. In: 2nd European Workshop on Model-Driven Architecture with Emphasis on Methodologies and Transformations. Volume Technical Report No. 17-04., University of Kent, Canterbury, UK (2004)
7. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: Ocl for the specification of model transformation contracts. In Patrascoiu, O., ed.: OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal, University of Kent (2004) 69–83
8. Vanhooff, B., Berbers, Y.: Supporting modular transformation units with precise transformation traceability metadata. In: ECMDA Traceability Workshop, SINTEF, (2005) 15–27

Composition of Model Differences

A. Cicchetti, D. Di Ruscio, and A. Pierantonio

Dipartimento di Informatica
Università degli Studi di L'Aquila,
67100 L'Aquila, Italy
{cicchetti,diruscio,alfonso}@di.univaq.it

Abstract. Difference calculation and subsequent representation are interesting and useful operations between models. Whilst the former can be performed by means of a number of tools, the latter is often based on case–specific solutions with limited abstraction. Therefore, a model representation for differences can be convenient both to depict information needed in a general way and to perform interesting operations such as parallel compositions. Starting from different concurrent versions of a certain model, represented as change documents, the whole spectrum of updates is obtained by merging such parallel changes. This paper illustrates how to deal with conflicting modifications of the same model elements by conceiving the composition of difference models as a particular kind of model transformation composition.

1 Introduction

Increasingly, complex software systems are cooperatively designed in distributed environments. The various kinds of design–level structural modifications a software system undergoes during its life–cycle can be detected for specifying differences in subsequent versions of design documents. The interaction among such modifications inevitably involves conflicts which must be reconciled. Taking advantage of current metamodel facilities, a difference model can be given according to a profile (introduced in [4]). In general, differences between model versions are mainly represented in operational terms or by coloring techniques (see [1, 3, 6]). However, leveraging the abstraction level is crucial to better comprehend the rationale behind the modifications and to harness the capabilities offered by modeling platforms. Interestingly, regardless of the calculation algorithm being used, the profile allows the result to be always represented with a model, although the difference is not necessarily a model (in a similar way that the difference between two natural numbers is not a natural number but a negative one). These models may represent different design versions of the same software system, which is possibly divided into several subsystems created in parallel by several designers on different platforms. In [4], the profile has been given a dynamic semantics and a difference model gives place to a specific kind of model transformation, i.e. specifying the evolution from an original to a final model by a difference model, then it can be applied to the original model to automatically obtain the final one.

This paper focuses on the composition of model differences, especially it proposes a domain specific language for specifying criteria to handle change conflicts raising from a parallel composition of modifications. The criteria can enhance the automatic composition of the differences by minimizing the manual intervention of the involved designers. Their intervention is expected only when the described criteria do not cover specific conflicts, or when manual and ad-hoc reconciliations are explicitly required.

The structure of the paper is as follows: Section 2 introduces the difference abstraction, showing the minimum information about syntax and semantics of the model. In Section 3, we specify an approach for weaving parallel changes, i.e. to merge concurrent deltas solving conflicts as described in the weaving itself. In Section 4 some related works are discussed. We conclude in Section 5 by stating what problems are solved

2 Specification of Model Differences

In [4] we proposed a UML profile to represent modifications of a model. The profile can be (logically) divided into three fundamental fragments dealing with the various kind of model modifications:

- addition and deletion of classes as a whole;
- structure update or moving of classes;
- addition, deletion and update of relationship between classes.

The first one is used whenever new classes are created or existing classes are deleted; in such cases the classes are marked with the stereotypes ≪added≫ and ≪deleted≫, respectively. The second part concerns updates to the class components, as attributes and methods, including the addition, modification, and deletion of them. Classes involved in such kind of differences are decorated with the ≪updated≫ stereotype and show only those elements which changed. Different versions of updated classes are linked by means of a directed association. Moreover, it is possible to represent shifts of attributes and methods from a source class to one or more target ones; in such cases the source is stereotyped with ≪moved≫ and contains all the elements involved. As in the update, an association links the source with all targets, which are named empty classes. Finally, the last fragment comprises all differences affecting relations between classes; it is possible to specify the addition, the deletion and the update of any relationship by using ≪added≫, ≪deleted≫ and ≪updated≫ stereotypes for the involved relations, respectively. In the latter case the relation is enriched with two tagged values, `from` and `to`, in order to denote modified properties (for example the multiplicity). Any kind of differences can be specified except those in diagrams with a layout semantics as sequence diagrams. See Fig. 1 for a simple example.

Finally, in order to avoid any ambiguity in the information conveyed in difference models since they are used throughout a tool chain, the profile is given formal dynamic semantics by means of Abstract State Machines (ASMs) as shown in [4].

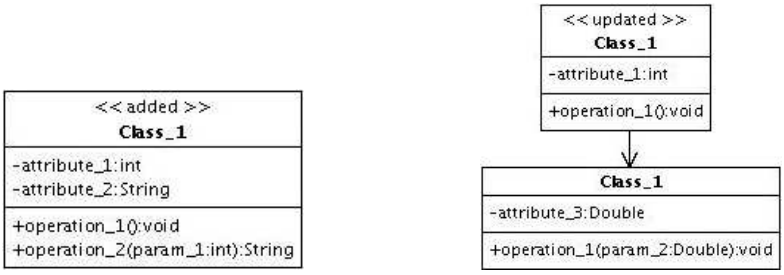

Fig. 1. Sample Difference models.

3 Parallel composition of difference models

In this section, we investigate the problem of conflicting modifications represented in distinguished change documents which have been produced in a distributed development environment (see Fig. 1). In particular, a Domain Specific Language (DSL) is proposed for specifying criteria to handle change conflicts, regardless algorithms to detect them, raising from a parallel composition of modifications. The criteria described by means of the proposed DSL can be evaluated by a resolving procedure able to perform the *parallel composition* [4, 12], that is the merging operation of the difference models.

By going into more details, the result of two parallel modifications can give place to conflicting results, i.e. elements in the original model which are changed by both difference models without converging to a common result. In this case, conflicting modifications have to be resolved by the corresponding designers (see for instance [1]). The different kinds of updates interact each other by eventually giving place to conflicts as illustrated in Table 1 where some combinations of differences (denoted by δ) and the related conflict possibilities between classes (or relations) are presented.[1]

δ_1 / δ_2	new	update	delete
new	YES	NO	NO
update	NO	YES	YES
delete	NO	YES	NO

Table 1. Some Difference combinations.

For example, two concurrent *new* actions give place to a conflict whenever they impose the same model element to have contradicting properties; the same problem can happen with two *update*. Other conflicts arise when the concomitant *update* and *delete* of a certain element are specified, then the changes must be reconciled by a negotiation among the corresponding designers. Finally, the remaining combinations are always not conflicting, as for instance when a new element in one modification cannot appear as updated or deleted into another, just because in the common ancestor, i.e. the initial version, it does not exist.

[1] For sake of space the table is not complete; in fact, it does not show interference situations between class and relation modifications.

As mentioned, conflicts cannot be automatically resolved when they have been caused by parallel updates of the same elements, then designers' intervention is needed; unfortunately, this activity is time–consuming and error–prone since it does not scale up well with the complexity of the software system. Besides, modification reconciliation is often guided by well–defined criteria set, heuristics and specific knowledge. Therefore, in the sequel a weaving model is described to explicitly represent such criteria and, consequently, to automate the parallel composition. Each different conflict, similar to the ones in Table 1, could be given corresponding resolution criteria; therefore in the proposed model should be depicted all possible conflicting couples in our differences representation. Each couple can appear in the weaving model linked by an undirected association, which is decorated with three tagged values, namely `appliedTo`, `rules` and `criteria`. The first tag can be valued `class`, `attributes`, `methods` or `attributes,methods` (default), and is used to modulate the granularity of the rules application, i.e. the class as a whole, only attributes, only methods, or both attributes and methods, respectively[2]. The second one specifies the rules which will be used in the criteria; for example `date:mostRecent` declares a rule named `date` which returns the latest element version. Finally, the third tag groups denote the desired criteria by means of rules and boolean operators, and it is used to drive modifications composition.

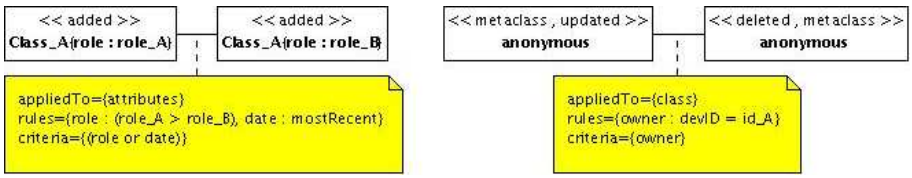

Fig. 2. Sample Weaving model.

To better explain the proposed model, a simple example is shown in Fig. 2, where the left part of the picture specifies the behavior with conflicts related to the addition of the class named `class_A`. The tagged value `appliedTo` narrows the criteria application to attributes, `rules` defines a predicate for roles and one for addition date priorities; finally, `criteria` combines the rules to drive the resolution. On the contrary, the right part is not related to a particular class, and for each pair of classes (≪updated≫, ≪deleted≫) prescribes to consider only those modifications which are owned by the developer whose id is id_A. In fact, `appliedTo` value is `class`, `rules` contains the predicate about owner identity which is used by `criteria`.

There are several points to remark: first of all, the criteria can be specified at several levels of granularity, as model level, element type level, or single element level. Thus, the resolving procedure starts from the finest granularity, forwarding the unsolved problems to coarser–grained definitions until the top level is reached. Moreover, each criteria can re–use rules described at upper levels and define its own rules, giving the weaving a high customization rate. However, it is

[2] It has to be noted that in the case of relationships this tag is ignored.

possible that some conflict cannot be reconciled, necessitating manual intervention; by knowing the designers who originated the update (by the tagged value `devID` associated to each model element), it is possible to automatically alert the ones involved in the situations, for instance, or post process the model to feed a workflow system.

Even if the proposed model is specifically developed to the parallel composition of change documents, several common aspects of transformation composition can be detected. In fact, such operation should solve possible conflicts between rules, and should use a flexible approach to minimize the manual intervention.

4 Related Work

There exists a number of works which define methods for detecting differences between two documents and only few focus on visualization and composition issues. In particular, the difference representation are often given in operational terms, i.e. structured documents, including UML models, can be represented as trees and differences can be detected by means of algorithms [2, 9, 11, 13] based on different set of atomic operations. All sets include the basic operations to create, delete or modify a node of the tree but only some of them can distinguish between the shift of a node and the combination of an insert and a delete. Thus, differences are given as sequence of such operations and are called edit scripts [3]. Another operational representation which presents similarities with edit scripts is given in [1] where visualization is given with sequence of operational terms. Another well-known visualization technique is coloring [5], where the original and final models are merged together distinguishing the various operations on the elements by means of different colors. In this paper, the differences are represented in a declarative way by means of suitable models that can enhance the interoperability between various modeling tools.

To the best of our knowledge, composition of differences is only partly considered in few works which mainly focus on software refactoring [10, 7] and differences occurring on systems developed in parallel [12]. This paper tries to raise the level of abstraction by reformulating part of the knowledges and experiences of these works in a Model Driven setting. In particular, the differences and their compositions are expressed by means of proposed modeling constructs and a conflict resolution mechanism based on priorities and roles is also provided. Finally, in [8] a number of operators for model integration are described and they have partially inspired the weaving constructs proposed in this paper.

5 Conclusions

In this paper, the problem of reconciling the conflicts which arise from composing change documents is discussed. Difference models are given as profiled UML diagrams in order to record the modifications a software system has been subject to during its life–cycle. Moreover, a differences model gives place to a specific kind of model transformation since the final model can be automatically obtained by

applying the differences model to the original model. Therefore, the composition of difference models is a particular kind of model transformation composition. The model transformation is defined by means of the dynamic semantics which operationally describes how to reconstruct a final model by starting from the original model and the differences model. In this paper, the reconciliation of the conflicts is defined according to criteria which can be given at different degree of granularity. In particular, the resolution starts from the finest granularity, forwarding the unresolved conflicts to coarser–grained definitions until the top level is reached. When specific conflicts do not reconcile the involved designers are required to negotiate the manual intervention. All the information is encoded in the models enabling the automated processing of difference models and their composition within a given tool chain. Further work is necessary to develop a tool to assist the proposed approach especially for supporting the criteria specification and the resolving procedure. The validation of the work against a realistic problem is also needed.

References

1. M. Alanen and I. Porres. Difference and Union of Models. In *UML 2003*, volume 2863 of *LNCS*, pages 2–17. Springer-Verlag, 2003.
2. D.T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree Correction for Document Trees. Technical report, Departement of Computing and Information Science Queen's University Kingston, Ontario, Canada, Jan 1995.
3. S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proc. ACM ICMD'96*, pages 493–504, 1996.
4. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Domain-Specific Modeling Language for Model Differences. Technical report, Dipartimento di Informatica, Università di L'Aquila, http://www.di.univaq.it/di/pub.php?page=1&id=728, 2006.
5. D.Ohst, M. Welle, and U. Kelter. Difference Tools for Analysis and Design Documents. In *19th Int. Conference on Software Maintenance (ICSM 2003)*, pages 13–22. IEEE Computer Society, 2003.
6. D.Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. In *ESEC/FSE-11: Proc. ESEC/FSE*, pages 227–236. ACM Press, 2003.
7. T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electr. Notes Theor. Comput. Science*, 127(3):113–128, 2005.
8. T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Model Integration Through Mega Operations. In *MDWE2005*, 2005.
9. S. M. Selkow. The tree-to-tree editing problem. *IPL*, 6(6):184–186, Dec 1977.
10. G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J. M. Bieman. Model Composition Directives. In *UML2004*, volume 3273 of *LNCS*, pages 84–97. Springer-Verlag, 2004.
11. K.-C. Tai. The Tree-to-Tree Correction Problem. *Journal of the ACM*, 26(3):422–433, 1979.
12. G. L. Thione and D. E. Perry. Parallel Changes: Detecting Semantic Interferences. In *COMPSAC*, pages 47–56. IEEE Computer Society, 2005.
13. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

Blackbox Composition of Model Transformations using Domain-Specific Modelling Languages

Dennis Wagelaar*

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
dennis.wagelaar@vub.ac.be

Abstract. In Model-Driven Engineering, multiple model transformations often have to be composed to produce a common result. One composition approach is to compose rules from existing transformations into one transformation, which generally requires the transformations to be written in the same language. This paper focuses on chaining model transformations together by having them pass models to each other, thus treating model transformations as blackbox components. Similarly to blackbox software components, not all model transformations can be combined. There are also constraints to the order in which model transformations have to be used. In the domain of component composition, Domain-Specific Modelling Languages (DSMLs) have been used to drive verification of compositions and automatic generation of composition code. We propose to apply DSML techniques for the composition of model transformations.

1 Introduction

There are many scenarios in Model-Driven Engineering in which a number of model transformations have to be composed in order to produce a common result. These scenarios range from one transformation language applied to one meta-model to many transformation languages applied to multiple meta-models. Each of these scenarios has different possibilities and limitations for composition. In some scenarios, it is possible to compose rules from existing transformations into one transformation. This generally requires the transformation rules to be written in the same language. This paper focuses on chaining model transformations together by having them pass models to each other, thus treating model transformations as blackbox components. This approach should be applicable in all scenarios and can be combined with the first approach.

Similarly to blackbox software components, not all model transformations can be combined. For the model transformations that can be combined, there are often constraints to the order in which they have to be applied. In the domain of component composition, Domain-Specific Languages (DSLs) have been used

* The author's work is part of the CoDAMoS project, which is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT-Flanders)

to drive verification of compositions and automatic generation of composition code [1]. In the case of a Domain-Specific Modelling Language (DSML) [2][3], the configuration rules and constraints are encoded in the meta-model of the DSML.

A DSL (or DSML) typically contains language constructs that refer to a particular application domain (e.g. web registration systems or instant messaging), whereas general-purpose languages use far more abstract language constructs and force the programs/models to fill in the concrete semantics.

We propose to apply DSML techniques, such as meta-modelling and transformation, for the verification of model transformation compositions and automatic generation of composition code. In the remainder of this paper, we will illustrate the use of meta-models and transformations for expressing composition rules and for generating build scripts that implement the composition.

2 Expressing Composition Rules

Using the Eclipse Modeling Framework [4], one can define a meta-model for a domain-specific composition language. As an example domain, a set of Java-specific refinement transformations is used. These transformations apply local refinements to model written in the Unified Modeling Language (UML) version 1.4 and use Java as an Action Language. The kinds of refinement include refining associations to attributes, attributes to accessor methods, observer stereotypes to a Java implementation, applet steretypes to a Java implementation and a few others[1]. Fig. 1 shows (part of) the meta-model for the "Java-refinement-specific" composition language.

The meta-model enforces the order in which the transformations must occur: each RefinementConfiguration must start with an AssociationAttributesRefinement. AssociationAttributesRefinement is an abstract meta-class, which has two concrete subclasses: AssociationAttributes and Java2AssociationAttributes. Note that the meta-model also enforces one to choose exactly one AssociationAttributesRefinement, since the *first* attribute of RefinementConfiguration has a multiplicity of *1*. The meta-model then goes on by enforcing that the *next* transformation after AssociationAttributesRefinement should be an AccessorsRefinement. This is again an abstract class with two concrete subclasses: Accessors and Java2Accessors. After AccessorsRefinement come ObserverRefinement, AppletRefinement and other that are no longer shown.

Note that there is another rule to the composition of AssociationAttributesRefinement and AccessorsRefinement: if one chooses to use AssociationAttributes, one has to choose Accessors. Alternatively, if one chooses to use Java2AssociationAttributes, one has to choose Java2Accessors. This is because both transformations use Java collection types to implement multiple value attributes and its accessor methods. The Java types used in both transformations have to be the same. In order to express this in the meta-model, the *next* attribute

[1] The transformations themselves can be found at http://ssel.vub.ac.be/ssel/research:mdd:casestudies

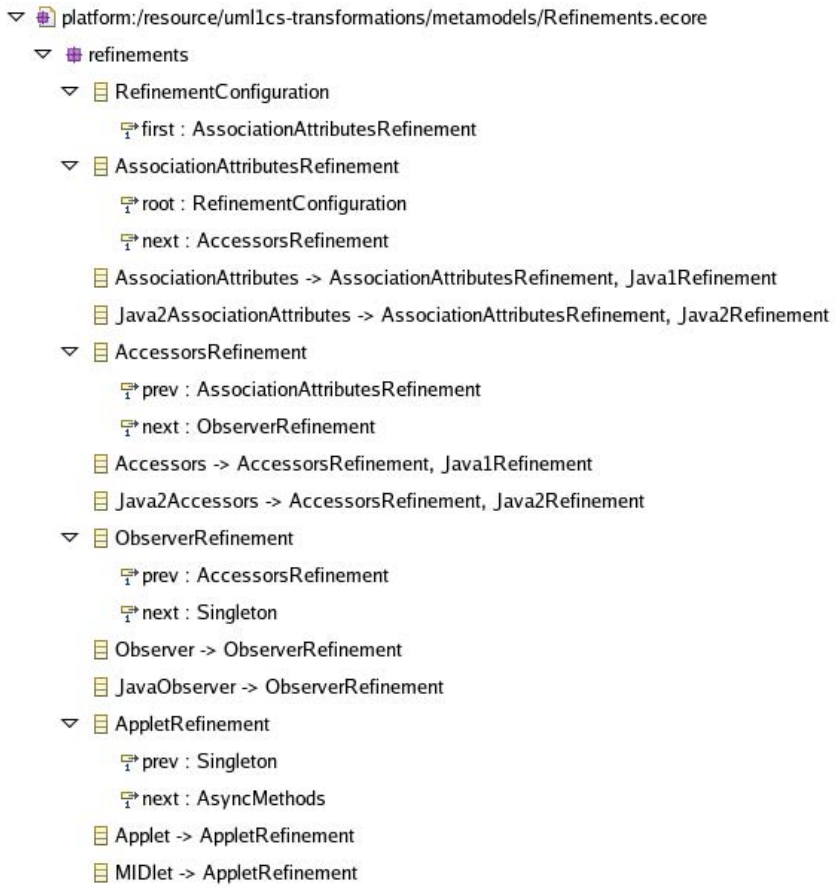

Fig. 1. Part of the EMF meta-model of the refinement model transformations

of AssociationAttributesRefinement would have to be pushed down to its subclasses and point to the correct AccessorsRefinement subclass. While this can still be done relatively efficiently for transformations that reside next to each other in the execution order, it would have been very cumbersome if AccessorsRefinement came after AppletRefinement, for example. It also tangles the order enforcing rules with the Java type consistency rules. In our example meta-model, we've chosen to introduce two new abstract meta-classes, Java1Refinement and Java2Refinement, which can be used in a model transformation that checks type consistency, such as the following model transformation written in ATL [5]:

```
rule Java1Java2 {
  from s : DSL!Java1Refinement (
    DSL!Java2Refinement.allInstances()->notEmpty())
  to t : REPORT!Error mapsTo s (
    message <- 'Java1 and Java2 refinements cannot be combined')
}
```

Note that this transformation rule does not need to know about any concrete transformations or the order in which they must be executed. It just checks for combinations of Java1Refinement and Java2Refinement instances.

3 Generating Composition Code

In addition to validation of transformation compositions, it is also possible to generate build scripts from transformation compositions. Since these transformation compositions are again models, adhering to the rules expressed in the meta-model (see Fig. 1), we can use model transformations to generate the build scripts. Below is a partial example ATL transformation that generates code for an Ant build.xml file:

```
query GenerateBuildFile = DSL!RefinementConfiguration->allInstances()
        ->collect(e|e->toString()->writeTo('build.xml'));

helper context DSL!RefinementConfiguration def : toString() : String =
        thisModule->header() +
        self.first->toString(input) +
        thisModule->footer();

helper def : header() : String =
        '<?xml version="1.0" encoding="UTF-8"?>\n' +
        '<project name="refinement" default="transform">\n' +
        '    <target name="transform" depends="clean">\n' +
        '        <atl>\n';

helper def : footer() : String =
        '        </atl>\n' +
        '    </target>\n\n' +
        '</project>';

helper context String def : atlRefineCommand(input : String, merge : String)
 : String =
        '<!-- ' + self + ' -->\n' +
        '<arg line="--trans ${transf.uri}' + self + '.asm"/>\n' +
        '<arg line="--in IN=' + input + '.ecore UML=${mmodel.uml} EMF"/>\n' +
        '<arg line="--in MERGE=' + merge + '.ecore UML=${mmodel.uml} EMF"/>\n' +
        '<arg line="--out OUT=' + input + 'r.ecore UML=${mmodel.uml} EMF"/>\n' +
        '<arg line="--lib Java=${lib.java}"/>\n';

helper context DSL!AssociationAttributes def : toString(input : String)
 : String = 'AssociationAttributes'->atlRefineCommand(
        input, '${rmodel.ocltypes}') + self.next->toString(input + 'r');

helper context DSL!Java2AssociationAttributes def : toString(input : String)
 : String = 'Java2AssociationAttributes'->atlRefineCommand(
        input, '${rmodel.ocltypes}') + self.next->toString(input + 'r');

helper context DSL!Accessors def : toString(input : String) : String =
        'Accessors'->atlRefineCommand(input, '${rmodel.ocltypes}') +
        self.next->toString(input + 'r');

helper context DSL!Java2Accessors def : toString(input : String) : String =
        'Java2Accessors'->atlRefineCommand(input, '${rmodel.ocltypes}') +
        self.next->toString(input + 'r');
...
```

4 Conclusion

This paper has illustrated how DSML techniques can be used to perform black-box composition of model transformations. A meta-model is used to express the rules for composition, along with any additional model transformations for validation. Since the transformation compositions are models themselves, it is possible to use model transformations for generating build scripts. Such build scripts can be used to implement the transformation composition.

In our example, the execution order of the transformations, which is already encoded in the DSML meta-model using the *next* and *prev* attributes, is explicitly navigated for each meta-class in this build script generator. With the current meta-model used, this cannot be avoided. If we would have modelled the *next* and *prev* attributes in a general superclass in the meta-model, the build script generator could have used one general helper method to navigate from one transformation to the next. Doing this would however also remove the execution order rules from the meta-model itself and an additional validation transformation is necessary to check the ordering.

From the above observation, it seems that the DSML meta-model should be mainly used as an enabler for validation and generation transformations. By using abstract meta-classes to model a kind of rule, such as "execution order" or "type consistency", external validation transformations can be used to check such rules while being oblivious of other rules. Similarly, build script generator transformations can be oblivious of the composition rules and only needs to navigate through the composition model.

References

1. Deursen, A.v., Klint, P.: Domain-Specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology **10** (2002) 1–17
2. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. IEEE Computer **34** (2001) 44–51
3. Tolvanen, J.P., Rossi, M.: MetaEdit+: defining and using domain-specific modeling languages and code generators. In: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003), Anaheim, CA, USA, ACM Press (2003) 92–93
4. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. The Eclipse Series. Addison Wesley Professional (2003)
5. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaice. (2005)

Aspects of Reusable Model Transformations

Gøran K. Olsen[1], Jan Aagedal[1], Jon Oldevik[1],

SINTEF Information and Communication Technology
Forskningsveien 1,
0373 OSLO, NORWAY

{Goran.K.Olsen, Jan.Aagedal, Jon.Oldevik} @sintef.no

Abstract. Model transformations are one of three pillars that OMG's Model Driven Architecture is built on. Together with metamodels and models it is a vital part. Today model transformations are mainly written from scratch for each problem they are to help solving. This is in many cases a very time consuming and difficult task. If the vision of MDA is to be fulfilled we believe that there must be large libraries of reusable model transformations available. This paper describes aspects of reusable transformations: It defines reuse in the context of transformations, identifies characteristics of reusable transformations, ways of achieving reuse and presents three studies conducted in the MODELWARE project.

1 Introduction

MDA emphasizes that models should be considered as first degree artifacts and be used and kept in sync throughout the software development process. The software systems should be designed at a high abstraction level and then incrementally refined to contain more specific and detailed information, ending at the implementation of the system in a given language. In MDA, there is a belief that automation of parts of these refinements would be beneficial, both increasing the productivity and the quality of the developed systems. An even greater gain could be achieved by enabling extensive reuse of these transformations. Reuse of transformations is an absolute requirement, since writing the transformations from scratch each time would be too time consuming, and the quality benefits of reuse would be lost.[1]

2 Transformations

A model transformation can be viewed as a transformation between two model spaces defined by their respective metamodels. A source model to target model transformation specification describes how elements in the source model space should appear in the target model space by relating metamodel elements in the source and target metamodels. The source model instance conforms to the source metamodel (for instance the UML2 metamodel). The transformation implementation transforms a

source model instance to the corresponding target model instance. The target model instance conforms to the target metamodel (for instance the UML profile for EJB). In this context we are interested in transformation specifications that can be reused.

3 Transformations Reuse

The idea and debate concerning reuse in software development have been present in the software community for decades:

> **Reusability.** The degree to which a software module or other work product can be used in more than one computer program or software system.[2]

Reuse has been touted as the silver bullet for many new technologies, e.g., object-orientation and component-based software engineering. Despite the hypes, reuse has been applied successfully in many different settings, spanning from the opportunistic approach of cutting and pasting code lines, to a more structured and systematic approach like creating and utilizing assets, for instance architectures, patterns, components and frameworks.

Reusing software assets involves several steps, such as: Identification, specification, storage, retrieval, application and assessment of costs and benefits. In order to cope with these challenges there is a need to understand how reuse applies to the transformation domain.

In this paper we discuss how to apply the results from the reuse domain to model transformations, making them reusable assets. An overall goal of the MODELWARE project is to increase the productivity of software development with 15 – 20 %, and we believe reuse of model transformations will significantly contribute to this. If the transformations must be written from scratch each time, this will be time consuming and one can expect that the cost of transformation development may sometimes even outweigh the benefits of applying them. Therefore one of MODELWARE's important results is to establish libraries containing reusable transformations. In these libraries it should be possible to search for and retrieve information on already existing transformations in a quick and simple way such that all necessary information is made available.

4 Characteristics of Reusable transformations

In order to characterize a reusable transformation, we need to characterize the properties that make a transformation reusable as opposed to just usable in a specific case. We define a reusable transformation to be *a transformation that can be used in several contexts to produce a required asset.* With different contexts we mean for instance different applications, projects, companies, or domains. In the following we discuss how the specificity of the source, target, and modeling element relationships influence reusability of a transformation.

4.1 Specificity

A model is always an abstraction of the system under consideration. This means that a model specifies certain aspects of a system, whereas other aspects are not considered. A model may therefore represent many systems, varying by the aspects not defined by the model. If one refines a model, one adds more detail. This means that one restricts the possible systems further, reducing the number of candidate implementations. The recursive process of refinement ends when there are no degrees of freedom and it therefore only exist one possible solution: the implemented system.

The level of specificity of a model influences its reuse potential. An abstract model can represent many systems, and is therefore reusable in many contexts. However, the knowledge embedded in a detailed model is much larger than in an abstract one, so the value of reusing a detailed model is much greater. In the following we discuss this trade-off in specificity, and address the source and target models, and the actual transformations.

Source and Target Model Specificity. The more details the metamodels contain, the more restrictions are put on the models that complies with these, and the less reusable are the transformations to which these models are source/target. It therefore is clear that the metamodels should be as small and little restrictive as possible while still containing the necessary information for the transformation specification.

There are several orthogonal dimensions of a modeling language: from general purpose to domain-specific, from high-level to low-level, and from focused (e.g., only process modeling) to complete (i.e., able to model all aspects of the systems). The number of contexts that a transformation fits in depends on how the metamodels position the modeling language on these dimensions.

Transformation Specificity. A transformation specification is in essence also a model, i.e., it specifies the relationships between source and target model elements. A complex transformation specification contains much knowledge, and the added-value of reusing such a transformation is therefore high. However, a complex transformation may make too many choices for the relationships between source and target, and it is therefore not possible to use in many contexts. One possibility to alleviate this is to divide a complex transformation into smaller transformations that can be sequenced. Using this approach, one may identify several reusable transformations even if the larger and more complex one is not reusable

A transformation specification can be made more general (and thereby more reusable) by parameterization. A transformation specification template can be instantiated to become many kinds of transformation specifications, depending on the value of the parameters used to instantiate it. For instance, a transformation specification template can take the source metamodel type as a template parameter and use this to resolve the specified mappings to actual metamodel properties of the source metamodel. This may be achieved by defining the transformation without referring to any specific properties of the source metamodel; for instance choosing a meta-metamodel such as MOF2.0 that defines some generic properties (e.g., name, isQuery) of all metamodels that can be used in the transformation specification.

5 Ways of reusing

There are several different approaches for making use of a reusable transformation: From the transformation point of view:

As-is. The simplest way of reuse is when a transformation is found in a transformation library and the specification fits the requirements. The transformation can be reused as-is without additional specialization. This is typically a very general transformation that generates common static structures or does some kind of re-factoring.

Another approach of reusing a transformation as-is is the creation of new transformations based on reusable transformation fragments. In this case, the developer uses fragments of a transformation as-is. We might see a scenario where a large amount of rules/operations are collected in a library in the same manner as what is the case with the reusable transformations.

Composition / Chaining. To be able to compose a transformation, it requires that several reusable transformations already exist and these transformations can be composed in such a way that they produce the required output. This means that the output of one transformation can be used as the input of another transformation. It also means that there is a way to define the composition, e.g., by defining a composite transformation that can invoke or otherwise include other transformations.[3]

Specialization. If a generic transformation needs to be extended to better fit the current needs, it may be possible to specialize it, e.g., by using inheritance. This might be a scenario where the original transformation is made very generic such that it will fit a larger number of problem domains. Instead of making your own from scratch you can continue building on an already stable and well tested transformation. You will also have the ability to import so called “utility” or “helper” transformations.

Parameterized. As previously described, a transformation specification can have parameters that need to be supplied when reusing it, either template or actual parameters. Note that parameterization can be combined with other reuse approaches such as specialization and composition.

Higher-order. A higher-order transformation is a model transformation that transforms other model transformations. A higher-order transformation uses the fact that a transformation itself is a model compliant with a metamodel, and one can therefore define transformations that take models of transformations as input (e.g., models conformant with the MOF 2.0 QVT metamodel[4]) and produces other transformation models as output.

Opportunistic. There will be a number of transformations that might not be applicable to more than a few applications. These and all the other transformations can be subject for opportunistic reuse (edit / copy / paste) and work as guidance for other similar problems.

Modify/filter the source model. As transformations are constrained by source and target metamodels, a way to enhance transformation reuse is to reduce the scope for each transformation input metamodel. This approach needs to filter the source model in source model subsets, define the necessary glue between compositions and finally combine the obtained target sub-models together.

6 Experience in practice

Several experiments with model transformations have been conducted in the MODELWARE project over the last years. Different tools and languages have been tested for reusability purposes and some of the experiences are presented in the following sections. The different ways of reusing a transformation explained in the previous sections are derived from these experiments.

ATL Transformations and library. ATL is the ATLAS INRIA & LINA research group answer to the OMG MOF/QVT RFP. Several transformations have been written in this language and some of them have been utilized as reusable transformations. The ATL project is part of the Eclipse GMT project and on this web site the transformations are published in a preliminary transformation library. Present the library contains almost fifty different transformations where some of them have been subject for some amount of reuse. There is also developed a transformation specification sheet that has given experience in how information about the different transformations should be structured[5-7].

J-language Transformations. At Thales there have been developed several transformations in the J-language which is the Objectering Profile Builder Tool's development language. There have also been developed a framework which is based on fine-grained unitary clone transformation rules for concept creation and fine-grained unitary ownership transformation rules for concept linking.[5]

MOFScript Transformations. MOFScript is a language and tool for writing model to text transformations. It is developed in the MODELWARE project by SINTEF. MOFScript is part of the Eclipse GMT sub-project and is also submitted to OMG as an answer to the MOF Model to Text Transformation RFP. A number of MOFScript transformations have been developed as examples, proof-of-concept and as part of cases. Most of the examples are based on the UML2 metamodel, however, there are also examples of using other metamodels, e.g. EMF and MOFScript's own metamodel. Reuse mechanisms supported in MOFScript includes rule polymorphism / overriding, transformation inheritance and transformation library imports (reuse by composition and invocation)[5, 8].

7 Conclusion

The experiments have shown that the importance of large libraries containing transformation can not be overstated. A major issue is sharing and retrieving the transformation in a way that makes reuse possible. Presently, the libraries do not have sufficient functionality, and need to be further developed.

We have seen some successful application of reusable transformations, mainly in the modeling domain (e.g. UML2Ecore) and with the use of fine-grained transformations that are being composed (chaining).

There are aspects of reusable transformations that make them more expensive to develop. More planning and analyses must be conducted. This will rapidly become economical defensible as soon as the transformations are reused.

In short, we conclude that there is not yet found any significant trace of large-scale reuse in the area of model transformation. This is not surprising, because the field is still very young. Our study warns, however, that unless we prove that some reusability of transformation software exists, the MDA approach is not likely to produce any economy of scale in the software development industry. As a consequence, we emphasize that the problem is serious and needs more investment than this study[5].

[1]**MODELWARE is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006). Information included in this document reflects only the author's views. The European Community is not liable for any use that may be made of the information contained herein.**

8 References

1. *OMG Model Driven Architecture,* http://www.omg.org/mda/. 2006.
2. *IEEE Standard Glossary of Software Engineering Terminology 610.12-1990. In IEEE Standards Software Engineering*. 1990.
3. Oldevik, J., *Transformation Composition Modelling Framework*. Vol. 3543 / 2005 2005: Springer Berlin / Heidelberg
4. OMG, *MOF QVT Final Adopted Specification* http://www.omg.org/docs/ptc/05-11-01.pdf. 2006.
5. MODELWARE, *D1.6 Definition of Reusable Transformations,* http://www.modelware-ist.org/index.php?option=com_remository&Itemid=79&func=fileinfo&id=77. 2006.
6. *ATL home page at:* http://www.eclipse.org/gmt/atl. 2006.
7. INRIA, *ATL: the ATLAS Transformation Language* http://www.eclipse.org/gmt/atl/doc/ATL_PresentationSheet.pdf.
8. *MOFScript Home page at:* http://www.eclipse.org/gmt/mofscript/. 2006.

Composing Transformation Operations Based on Complex Source Pattern Definitions

Arda Goknil[1], N. Yasemin Topaloglu[2]

Department of Computer Engineering, Ege University, Izmir, Turkey
[1]arda.goknil@ege.edu.tr, [2]yasemin.topaloglu@ege.edu.tr

Abstract. Rule composition and decomposition is a hot research topic within the context of model transformation. Mostly, transformation rules are considered as atomic parts of the transformation and rule composition has been the focus of recent research in the model transformation area. In our approach, we consider the transformation operations such as add, delete and update operations as the atomic parts of the transformation and the synthesis of these operations constitutes a single transformation rule. Defining complex and hierarchical source pattern definitions requires approaches and techniques about the composition and decomposition of these operations. In this paper, we discuss the problem statement and present an example case in which operation composition is required.

1 Introduction

Rule-based model transformation languages are the core technologies for operating the transformations between models on different abstraction levels in current *Model Driven Engineering (MDE)* approaches, such as *Model Driven Architecture (MDA)* [4] and *Model Integrated Computed (MIC)* [6]. In these languages, transformation rules are considered as the atomic elements of the transformation process.

Implementing large and complex transformations require complex and hierarchical pattern definitions to query models. In this context, complex pattern definitions mean that the pattern elements are tightly coupled and the relations between them are derived from the domain, not from the meta associations of the source metamodels. The coupling between the model elements is defined in the problem domain instead of metamodels. For instance, the relation between the *UML Class* and *UML Attribute* model elements in a proposed *UML2JAVA* transformation is derived from the UML metamodel. Complex source patterns include variation points and coupled elements hierarchically. Transformation rules which query these models should contain multiple operations and multiple pattern elements. In our approach, we consider the transformation operations such as add, delete and update operations as the atomic parts of the transformation and the collaboration between them constitutes a transformation rule.

In this paper, we highlight the need of transformation languages that support operation composition for complex pattern definitions. The paper is organized as follows. In Section 2, we discuss the transformations with operation definitions. In

Section 3, we present a sample transformation for composing transformation operations. Section 4 includes the conclusions.

2 Transformations with Operation Definitions

In transformation between two different metamodels, rules have simple definitions for transforming one model element in the source model into one or more model elements in the target model. This approach is called *one-many mapping*. In [7], the term *mapping* is defined as a synonym for correspondence between the elements of two metamodels, while the *mapping specification* precedes the *transformation definition*. Especially, transformation platforms which combine weaving and transformation, execute the transformations with one-many mapping. In transformations generated by mapping two different metamodels, the model elements are loosely coupled and the relations between them are derived from the source metamodel. The implicit rule calls defined in [1] solve the problem about integration and execution order of mapping rules. Composition approaches are mainly concerned about rule composition.

Since entities and the relations between these entities in the pattern definition are derived from the problem domain, they constitute the *hierarchical complex source patterns*. The rule structure requires complex pattern mapping within a single rule instead of one-many pattern element mapping within multiple rules. For instance, the relations between the pattern elements are defined by the problem domain in a proposed multiple inheritance-single inheritance (MI2SI) transformation [2] as a part of UML2JAVA transformation. In such a transformation operated by one-many mapping, the mechanism needs helper rules to define relations between the elements of the pattern. These helper rules make the problem definition more complex and incomprehensible. Transforming by pattern mapping takes the source pattern and transforms it into the target pattern by using less number of rules than one-many model element mapping uses. However, in this case, rules need multiple operations to transform one pattern into another one. Multiple operations in a transformation rule are given by Figure-1.

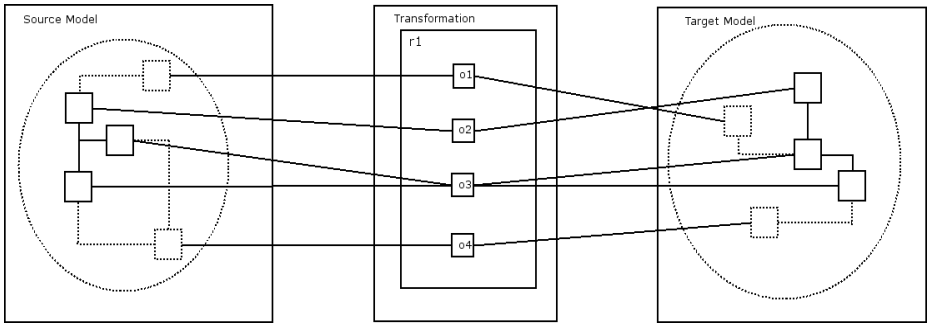

Figure-1. Multiple Operations in a Single Transformation Rule.

There are some issues to be considered in transforming by pattern mapping. Like rule composition, operation composition is required to manage and to organize the transformation operations defined between the complex pattern definitions. In Figure-1, the *r1* rule has four operations named *o1*, *o2*, *o3* and *o4*. The organization of the operations in the rule can be considered within the parallel of rule organization. The same problems about rule integration and organization occur in the operation organization and integration. There must be implicit operation calls in the rule to collaborate operations according to the relations between the pattern elements.

3 A Sample Transformation for Composing Transformation Operations

In this section, we explore the cases derived from the problem statements depicted in Section 2 over a sample pattern definition of multiple inheritance for a proposed multiple inheritance (MI) to single inheritance (SI) transformation. Although some programming languages include only single inheritance, defining a class by inheriting from more than one class is needed in a system design frequently. We consider UML models as the platform independent models which support multiple inheritance and Java programming language as our platform specific model which supports only single inheritance. The *MI2SI* transformation is a good example for complex pattern definitions because both source and target patterns contain multiple hierarchies and the transformation has multiple add, delete and update operations between MI and SI.

We consider the two alternatives as the representative cases of multiple inheritance (MI) since they can constitute the basis to generate other MI cases. In the first case, the inheritance hierarchy is composed of only one level and it is the simplest case of MI. The second case adds one more level to the inheritance hierarchy and called “diamond inheritance” [5].

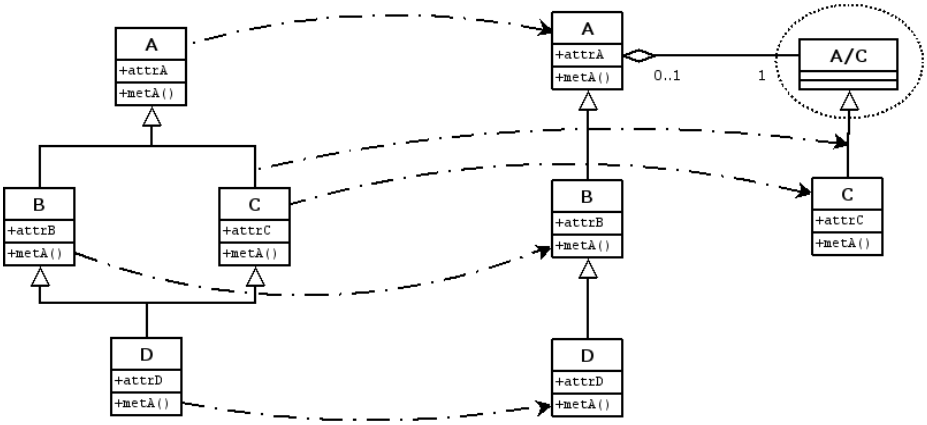

Figure-2. Transforming Multiple Inheritance to Single Inheritance with Role Aggregation.

In the hierarchy of multiple inheritance, we named the classes as *GrandParent*, *Parent* and *Child* classes. The *GrandParent* class is at the top of the hierarchy and the *Child* class is at the bottom. The *Parent* classes are the middle level classes which are the subclasses of the *GrandParent* class and super classes of the *Child* class. Figure-2 depicts the *MI2SI* transformation with role aggregation given in [2]. In Figure-2, one of the inheritance links is replaced by an aggregation link. The newly added abstract class named *A/C* is called *AbstractDiscriminatedClass* and the *Parent* classes in the *MI* which are the subclasses of the *A/C* class are now called the *ConcreteDiscriminatedClass* instead of *Parent* class.

The problem here is how to identify the rules between these two complex pattern definitions in the transformation definition. Decomposition satisfies the requirements for this identification. Kurtev [3] defined some rule decomposition approaches according to the target and source pattern. In source-driven approach [3], rules for every model element in the source pattern are defined. The rules for the *MI2SI* transformation are shown below:

GrandParentRule (**Source**[GrandParentClass],
Constraint[GrandParentClass, ParentClass, ChildClass],
Target[GrandParentJClass, AbstractJClass])

ParentRule (**Source**[ParentClass],
Constraint[ParentClass, GrandParentClass, ChildClass],
Target[ParentJClass, ConcreteDiscriminatedJClass])

ChildRule (**Source**[ChildClass],
Constraint[ChildClass, GrandParentClass, ParentClass],
Target[ChildJClass])

This transformation contains three rules. Each rule defines a source model element in its source part but each rule has the full definition of constraints to query the whole source pattern in the model. For instance, the *GrandParentClass* in the source part of *GrandParentRule* needs the full constraint definition of the source pattern to match in the model because the constraint part requires constraints of other source pattern elements related to the *GrandParentClass* to bind the appropriate model element. The helper rules are required in the constraint part to define the relationships between the pattern elements. In the *GrandParentRule*, we need to call two helper rules for the relation between the *GrandParentClass*, *ParentClass* and the *ChildClass*. The same helper rules and constraint repetitions are required for other rules named the *ParentRule* and the *ChildRule*. This kind of rule decomposition makes the definition more complex. We chose the composition of rules according to the source and target pattern mapping instead of rule decomposition.

MI2SIRule (**Source**[GrandParentClass, ParentClass, ChildClass, GPFeature],
Constraint[GrandParent, Parent, Child, GPFeature],
Target[GrandParentJClass, ParentJClass, ChildJClass,
ConcreteDiscriminatedJClass, AbstractDiscriminatedJClass,
GPJFeature, CDJFeature])

MI2SIRule maps the multiple inheritance source pattern and the single inheritance target pattern. The source and target parts of this rule include all pattern elements and the constraint part of the rules defines all relation and cardinality constraints of these pattern elements at once. In this kind of rule structure, we need to define the transformation operations which are the atomic parts of the transformation definition. There is a need of operation composition to organize and manage the appropriate operations within a single rule. Transformation languages should also support explicit definition of operation structures in rules. Every single operation should be able to map a number of source model elements to a number of target model elements.

MI2SIRule (**Source**[GrandParentClass, ParentClass, ChildClass],
Constraint[GrandParent, Parent, Child],
Target[GrandParentJClass, ParentJClass, ChildJClass,
ConcreteDiscriminatedJClass,AbstractDiscriminatedJClass]
Operation1[Type: Add, GPP_Generalization],
Operation2[Type: Delete, PC_Generalization],
Operation3[Type: Add, AbstractDiscriminatedJClass],
Operation4[Type: Add, GPA_Aggregation],
Operation5[Type: Add, AC_Generalization])

Another issue in the operation composition is the reuse of transformation definitions. *OMI2SIRule* depicts the transformation from one level multiple inheritance to single inheritance. Source pattern elements of the *MI2SIRule* except the *GrandParentClass* constitute the pattern elements of *OMI2SIRule*.

OMI2SIRule (**Source**[ParentClass, ChildClass],
Constraint[Parent, Child],
Target[ParentJClass, ChildJClass]
Operation1[Type: Delete, PC_Generalization],
Operation2[Type: Add, ParentFeaturetoChild])

We must decompose the constraint part of the rule structure to reuse the pattern elements and constraints of the *MI2SIRule* in the *OMI2SIRule*. The composed constraint structure of the *MI2SIRule* prevents the reuse of constraints for the *ParentClass* and *ChildClass* pattern elements. *MI2SIRule2* is the rule whose constraints are decomposed for every pattern element in the source part of *MI2SIRule*.

MI2SIRule2(**Source**[GrandParentClass, ParentClass, ChildClass],
ConstraintGP[GrandParent], **ConstraintP**[Parent],
ConstraintC[Child],
Target[GrandParentJClass, ParentJClass, ChildJClass,
ConcreteDiscriminatedJClass,AbstractDiscriminatedJClass]
Operation1[Type: Delete, GPP_Generalization],
Operation2[Type: Delete, PC_Generalization],
Operation3[Type: Add, AbstractDiscriminatedJClass],
Operation4[Type: Add, GPA_Aggregation],
Operation5[Type: Add, AC_Generalization])

As shown briefly in the above example, decomposing a transformation into operations and composing operations according to the complex pattern mapping make the transformation definitions more expressive. In addition, reusing the operations to constitute new rules is trivial.

4 Conclusion

In this paper, we discuss the need of operation composition and the general operation structure in transformation rules that transformation languages should support. Transformation languages should have additional features in their rule structures that provide operation composition. Composing operations within transformation rules will enable us to query and transform complex pattern definitions in a more expressive way.

References

1. Czarnecki, K., Helsen, S. Classification of Model Transformation Approaches. OOPSLA2003 Workshop on Generative Techniques in the Context of MDA, USA, 2003
2. Dao, M., Huchard, M., Libourel, T., Pons, A., Villerd, J.: Proposals for Multiple to Single Inheritance Transformation. In Proceedings of the 3rd International Workshop on Mechanisms for Specialization, Generalization and Inheritance MASPEGHI 2004 (Workshop ECOOP 2004), Oslo Norway, June 2004
3. Kurtev, I.: Adaptability of Model Transformations, PhD Thesis, University of Twente, 240p, ISBN 90-365-2184-X
4. OMG: MDA Guide Version 1.0.1. The Object Management Group, Document Number: omg/2003-06-01 (2003)
5. Sebesta, R.: Concepts of Programming Languages. Addison-Wesley Publishing, 2002.
6. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. Computer, Apr. 1997,pp. 110-112
7. Lopes, D., Hammoudi, S., Bezivin, J., Jouault, F.: Mapping Specification in MDA: From Theory to Practice. INTEROP-ESA'2005

Pattern composition in graph transformation rules ⋆

András Balogh and Dániel Varró

Department of Measurement and Information Systems
Budapest University of Technology and Economics
H-1117 Magyar tudosok krt. 2, Budapest, Hungary
{balogh,varro}@mit.bme.hu

Abstract. Graph transformation (GT) frequently serves as a precise underlying specification mechanism for model transformations within and between modeling languages. However, composability of graph transformation rules is typically limited to inter-level rule composition (i.e. rules calling other rules). In the current paper, we introduce intra-level composition for GT rules where the left-hand side and right-hand side graphs of GT rules can be composed of stand-alone graph patterns in a fully declarative way. As a result, the specification of complex model transformation problems can be drastically reduced. Our concepts are demonstrated using the transformation language of the VIATRA2 framework.

1 Introduction

Graph transformation (GT) [5], which provides a rule and pattern-based specification paradigm for the manipulation of graph models, is frequently used for specifying model transformations within and between modeling languages. When executing a GT rule on an instance model, a matching of the left-hand side (LHS) graph pattern is substituted by an image of the right-hand side (RHS) pattern. Since the early 1990s, a wide range of tool support has become available, e.g. PROGRES [9], FUJABA [7], AGG [6], GReAT [8], ATOM3 [4], TefKat [1], VIATRA2 [2].

On the one hand, graph transformation tools are able to handle industrial size models [10] for practical model transformation problems. However, GT specifications can easily become large and scattered in case of complex model transformations due to the limited support of composability in graph transformation languages. Most typically, the graph patterns in GT rules become large (consisting of a large number of pattern elements), and have often common partitions reused in many pattern.

In existing GT tools, composability is either completely missing (e.g. in AGG, or in ATOM3), or it is mainly limited to inter-level rule composition where a GT rule can be called from other GT rules, or by external control structures (as in PROGRES, FUJABA and GReAT), or the composition is similar to the object/oriented inharitance, where a rule can extends an other rules behaviour (like in case of TefKat). However, path expressions and multi-objects are the only mechanism which aim at compacting the definition of GT rules. Furthermore, while most of these tools offer support for defining

⋆ This work was partially supported by the European IPs SENSORIA and DECOS. The second author was also supported by the J. Bolyai Scholarship.

constraints (either in OCL or by graph constraints), the constraints and transformation rules are defined in different ways.

In the current paper, we introduce intra-level composition for GT rules in a fully declarative way. We first define graph patterns as a stand-alone and reusable specification concept, and then construct the LHS and RHS of GT rules by composing these predefined and also local patterns. As a result, the specification of complex model transformation problems can be drastically reduced. Our proposal also supports the reusability of GT patterns and rules by allowing the creation of predefined pattern libraries for typical GT steps. We demonstrate our concepts using the transformation language of the VIATRA2 framework.

2 Graph Transformation Rules in VIATRA2

The main transformation language of the VIATRA2 framework (called VTCL: Viatra Textual Command Language) [2] combines the declarative paradigm of graph transformation with the imperative constructs of the Abstract State Machines (ASM) [3] where elementary model transformation steps are carried out by graph transformation rules while complex transformations can be driven by ASM programs. While this approach is feasible form many model transformations in practice, the specification of fully declarative transformations frequently results in complex transformation rules.

The following code illustrates a simple graph transformation rule in VIATRA2, which operates on UML models and lifts up attributes from the child class to the parent.

```
gtrule liftAttrsR(in CP, in CS, in A) = {
  precondition pattern lhs(CP,CS,A,Attr) = {
        UML.Class(CP);
        UML.Class(CS);
        UML.Class.parent(Par,CS,CP);
        UML.Attribute(A);
        UML.Class.attrs(Attr,CS,A);
  }
  postcondition pattern rhs(CP,CS,A,Attr) = {
        UML.Class(CP);
        UML.Class(CS);
        UML.Class.parent(Par,CS,CP);
        UML.Attribute(A);
        UML.Class.attrs(Attr2,CP,A);
  }
}
```

The precondition pattern defines the LHS, and the postcondition pattern defines the RHS of the transformation step. When applying the rule, the pattern variables are substituted according to the matching, and these bound variables are passed to the RHS, where the difference of the RHS and LHS is calculated along the matching (i.e. edge attrs is removed between child class CS and attribute A, and a new edge attrs is created from parent class CP to attribute A).

In addition, VIATRA2 also supports negative application conditions defined as negative patterns, which may prohibit the application of the GT rule on a certain matching.

3 Composition of Graph Patterns in GT Rules

Now we discuss how graph transformation rules can be assembled by composing and reusing predefined graph patterns in the LHS and RHS of rules.

3.1 Composition of simple patterns

Now we can recognize two patterns that can be used at various places during UML model refactorings: the first is the class-attribute pattern, and the second one is the parent class-child class pattern. If we refactor the transformation by extracting these patterns we get the following code:

```
pattern classAttribute(C,A) = {
        UML.Class(C);
        UML.Attribute(A);
        UML.Class.attrs(Attr,C,A);
}
pattern parentClass(Parent,Child) = {
        UML.Class(Parent);
        UML.Class(Child);
        UML.Class.parent(Par,Child,Parent);
}

gtrule liftAttrsR(in CP, in CS, in A) = {
  precondition pattern lhs(CP,CS,A,Attr) = {
        find parentClass(CP,CS);
        find classAttribute(CS,A);
  }
  postcondition pattern rhs(CP,CS,A,Attr) = {
        find parentClass(CP,CS);
        find classAttribute(CP,A);
  }
}
```

The find construct is used for *pattern composition*, which means (in the simple case) the inclusion of other patterns in the current one by appropriate copying and renaming. Using this functionality we can compose our graph transformation rule using a predefined pattern set.

Simple *pattern composition in the LHS* of a GT rule means that all elements in the component patterns (after an appropriate merging) have to matched. Simple *pattern composition in the RHS* means that the merged pattern will define the overall RHS which defines the result of rule application.

Since both internal patterns (which reside in the current ASM machine) and external patterns (which reside in different ASM machines) can be called transparently using the find construct, this way one can create stand-alone pattern libraries, which are reusable from any other transformations. The reusability of patterns will require the presence of a developer documentation that provides a textual description of reusable components (like in many modern programming languages).

3.2 Composition of complex patterns

In case of the LHS of GT rules, the find construct also allows recursive pattern calls (the pattern calls itself), and OR-patterns which have multiple bodies (where a match of at least one body has to be found for a successful pattern matching).

The following rule illustrates both of these concept by defining the (transitive) ancestor of a class. The first part states that a parent is ancestor of a child if there is a direct parent relation between them, the second states that class Parent is ancestor of class Child, if there is a class C that is child of Parent and ancestor of Child.

```
pattern ancestorOf(Parent,Child) = {
        find parentClass(Parent,Child);
} or {
        find parentClass(Parent,C);
        find ancestorOf(C,Child);
}
```

When using such complex patterns in the RHS of GT rules, a new problem arises during execution. If the LHS contains recursive calls, the RHS should also be executed for all of the pattern matches. This requires the maintenance of the pattern call hierarchy (the execution stack of the LHS) and the execution of the RHS for all calls (by appropriate variable (re)naming). To illustrate the problem the following VTCL code introduces a pattern that can be used to lift the attributes of a class to all of its parents.

```
gtrule liftAttrs2R(in CP, in CS, in A) = {
  precondition pattern lhs(CP,CS,A,Attr) = {
        find ancestorOf(CP,CS);
        find classAttribute(CS,A);
  }
  postcondition pattern rhs(CP,CS,A,Attr) = {
        find ancestorOf(CP,CS);
        find classAttribute(CP,A);
  }
}
```

The execution environment needs to recognize that we have recursive patterns in both the LHS and RHS, and it has to “execute the RHS” for each level of the recursive call hierarchy of the LHS (and with the same pattern bodies in case of multi-body patterns). Such a recursive execution is sketched in Fig. 1.

4 Conclusions

In the paper we discussed composition mechanisms for graph transformation rules by merging predefined graph patterns both in the LHS (precondition) and RHS (postcondition) of the rules.

These mechanisms are already integrated in the VTCL *model transformation language* of the VIATRA2 framework. The transformation engine of VIATRA2 already has a stable implementation for the composition of both simple and complex patterns

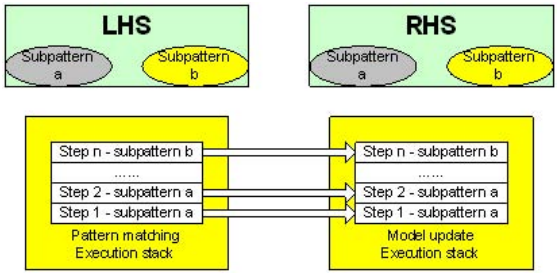

Fig. 1. Execution stack for recursive patterns in GT rules

in the LHS, while the implementation for pattern composition in the RHS is currently in an experimental phase.

Our practical transformation experiences in various projects show that the specification of complex model transformation problems can be drastically reduced (already if pattern composition is only supported for the LHS).

References

1. TefKat Project home page. `http://www.dstc.edu.au/tefkat`.
2. A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, pp. 1280–1287. ACM Press, Dijon, France, 2006.
3. E. Börger and R. Stärk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
4. J. de Lara and H. Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber (eds.), *5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings*, vol. 2306 of *LNCS*, pp. 174–188. Springer, 2002.
5. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.
6. C. Ermel, M. Rudolf, and G. Taentzer. *In [5]*, chap. The AGG-Approach: Language and Tool Environment, pp. 551–603. World Scientific, 1999.
7. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. Theory and Application to Graph Transformations (TAGT'98)*, vol. 1764 of *LNCS*. Springer, 2000.
8. G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 2003.
9. A. Schürr, A. J. Winter, and A. Zündorf. *In [5]*, chap. The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.
10. G. Varró, A. Schürr, and D. Varró. Benchmarking for graph transformation. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 05)*, pp. 79–88. IEEE Press, Dallas, Texas, USA, 2005.

Transformation Composition in QVT

Mariano Belaunde[1]

[1] France Telecom Recherche et Développment, 2 avenue Pierre Marzin,
22307 Lannion, France
mariano.belaunde@francetelecom.com

Abstract. This paper describes the mechanisms defined in the QVT-Operational [1] formalism to express composition of transformations. In particular it discusses the different granularities of composition and the effect on interoperability.

Keywords: QVT, MDA

1 Introduction

Transformation composition can be considered at two levels: coarse-grained composition is the capability to combine several *complete* transformations whereas fine-grained composition is the capability to combine *partial* elementary transformations – commonly called transformation rules. Typically, the former kind operates on *models* whereas the later operates on *model elements*.
The QVT-Operational formalism, which is part of the adopted OMG MOF 2.0 Q/V/T specification [1], provides, first of all, a set of elementary imperative constructs to express *chaining* of large transformations, which could, potentially, be implemented in different languages. In the other hand, it provides various high-level constructs to express fine-grained coupling of rules to build a complex composite transformation.

Section 2 of this paper examines the mechanisms that are available in QVT for coarse-grained composition and the following Section 3 examines the facilities that are available for fine-grained composition of rules.

2 Expressing coarse-grained composition in QVT

A command scripting language like *shell* in Unix systems allows to chain commands that can be implemented in different languages. The shell script languages offers the possibility to write loops, if/the/else controls, to manipulate variables, to pass string parameters to the commands and the possibility to retrieve the output of a command and to pass as input to the consequent command.
The ability to chain transformations is indeed a major scenario in MDA. There are excellent reasons for having transformations specified or implemented using different frameworks and different languages. One possibility we had when defining the QVT-Operational formalism was to let the chaining of large transformation out of the

standard and let this task to existing scripting languages as the shell. This remains of course possible but, as we will see, having in QVT the capability to express directly this coarse-grained composition has several advantages.

In essence, a QVT operational transformation declares a name, a signature stating the expected input models and the output models, an entry operation and a list of queries and mapping operations. A black-box transformation only declares a signature and serves to encapsulate a transformation implemented using an external language.

The example below could be the signature of a transformation that would generate a RDF model from a UML model and, as side effect, insert annotations in the UML model.

```
transformation Uml2rdf (inout umodel:UML,out rmodel:RDF);
main() { … }
```

Now let's imagine this *uml2rdf* is a composite transformation that chains two transformations: an in place *uml2annotateduml* that performs intensive analysis and produces automatic annotation to prepare the following step *annotateduml2rdf*, which is the production of the RDF model. The first transformation could be a transformation written directly in Java while the second one could be a QVT transformation. The code for the composite transformation will be something like:

```
transformation Uml2rdf (inout srcmodel:UML,out destmodel:RDF)
   access Uml2annotateduml(UML), Annotateduml2rdf(UML,RDF);
main() {
   var t1 := new Uml2annotateduml(srcmodel);
   if (t1.transform().failed()) then do {
       log("Failed UML transformation",t1.status);
       return;
   };
   var t2 := new Annotateduml2rdf (srcmodel,destmodel);
   if (t2.transform().failed()) then
       log("Failed RDF transformation",t2.status);
}
```

As demonstrated by this example, an explicit invocation of a transformation implies two steps: firstly the transformation is instantiated using the **new** operator, and then the transformation is invoked through the **transform** operation. Because it is a parameter with *out* direction, the model denoted by *destmodel* - representing an RDF model - is implicitly created and initialized before entering the entry point of the composite Uml2rdf transformation. The result type of the transform operation is a *Status* object storing a code status of the execution (like *fail* or *success* flag). .

The body of the entry operation contains a sequence of expressions. In this case we take advantage of imperative constructs in QVT to express in a simple and understandable way[1] the composition, which indeed, includes the production of arbitrary log messages when errors are encountered. Others typical constructs like OCL [2] collect and select operators or imperative *forEach* and *while* loop can be exploited here. To summarize, QVT has enough "programmatic" support to allow expressing arbitrary logic for chaining the transformations.

[1] An OCL expert would probably be capable of expressing the same thing using a unique long expression. But such a complex expression might be not of the taste of most users.

The main observation we can do at this level is that for coarse grained transformations QVT is not inventing something significantly new. It is simply using the same basic programmatic constructs that are available for fine grained composition of rules (if-then-else, loops, variables and so on) and that will be typically found in a scripting *shell* language. An interesting implication of this is that, for the purpose of composing coarse-grained transformations, there is no need for the user to "jump" into a different formalism[2].

If we continue the comparison with the *shell* language, there is however a noticeable difference regarding the type of parameters that can be passed to the "command" – a transformation in our case. Rather than simple strings, we are dealing here with *model types*, which basically are *MOF extents* with additional constraints implied by the MOF metamodel associated to the model type. This means that a tool implementing the language is expected to perform type analysis on models[3]. The notion of *model type*, referring indirectly to a metamodel, gives some flexibility in the kind of input models that can be accepted by a transformation. For instance, the UML model type required by the *uml2rdf* may be defined in a way that makes it possible to use both UML 1.4 or UML 1.3 conformant models. The flexibility on the model parameters enhances the possibilities for reusing existing transformations in composite transformations.

2.1 Parallelizing invocation of transformations.

Within a composite transformation it may be useful to indicate somehow that some of the composed transformations can run in parallel. This can be very important from the point of view of optimization, especially when dealing with large amounts of data as the input of a transformation. This facility is indeed available in standard command script languages like the shell, with the facility to execute commands in background. Less common in scripting language is the ability to synchronize upon the completion of a command executed in 'background'. In QVT-Operational, parallelism is expressed through the usage of a variant of the 'transform' operation: it is the 'parallelTransform' whose effect is to – potentially – create a new thread of execution and return immediately. The synchronization uses a **wait** operation on the status variable assigned when invoking the transformation in parallel.

The example below illustrates the usage of this facility. The Uml2rdf has been changed in suach a way that it performs now a merge of two annotated UML models. The *Uml2annotateduml* transformation is executed for each UML source model.

```
transformation Uml2rdf (inout src1:UML, inout src2:UML,out destl:RDF)
   access Uml2annotateduml(UML), Annotateduml2rdf(UML,RDF);
main() {
   var status1 := new Uml2annotateduml(src1).parallelTransform();
   var status2 := new Uml2annotateduml(src2).parallelTransform();
```

[2] The implicit assumption here is that there is - probably - no great added value to introduce specific concepts to express *chaining* of large transformations. This can be, of course, matter of discussion but there is no sufficient room in this paper for this.

[3] Various strategies for model checking are possible. The standard QVT defines only two pre-defined model type compliance kinds (strict and effective) but leaves open the possibility to define other variants.

```
    status1.wait();
    status2.wait();
    if (status1.failed()) then do {
        log("Failed UML transformation on first model",t1.status);
        return;
    };
    if (status2.failed()) then do {
        log("Failed UML transformation on second model",t1.status);
        return;
    };
    var t := new Annotateduml2rdf (src1,src2,destmodel);
    if (t.transform().failed()) then
         log("Failed RDF transformation",t2.status)
}
```

3 Expressing fine-grained composition in QVT

For composing elementary transformation rules the basic imperative constructs – if-then-else, loops, variables and so on – that are available for chaining coarse-grained transformations can be used. However, QVT defines specific high-level mechanisms to enhance the modularity and reuse of the language in situations where a simply chaining of rules is not sufficient for composition. The specific compositions facilities described from Section 3.2 to Section 3.4, introduces, at some extent, a declarative "look and feel" in the language – while remaining, in terms of the execution semantics, hopefully, totally imperative.

A typical use-case of fine-grained composition is when a transformation designer wants to build a transformation that reuses rules from an existing transformation already written, in order to specialize it or in order to apply the transformation in a slightly different context.

3.1 Object oriented background of QVT-Operational

Writing transformations may be a complex task and the nature of such a task is not so different than the task of building a complex program. QVT-Operational exploits well-proven object-oriented techniques: Transformations may inherit from other transformations, operations may be overridden and dynamic binding semantics applies on overridden operations. In addition QVT offers an extension mechanism which allows defining helper operations, mappings operations and intermediate properties as "logical" additional features of the pre-existing meta-classes involved in the transformation. The specific composition facilities described in the following sub-sections are defined in the context of this object-oriented background.

3.2 Guards in mapping operations

A mapping operation may declare explicitly a condition expression introduced by the "when" keyword. Depending on the invocation mode (**xmap** or **map**) this expression will be interpreted either as a simple pre-condition (raising an error if not

satisfied) or as a guard (the body of the operation is not invoked, null is returned instead).

The guard facility will have an effect on the way mappings operations are composed since the testing expression for invoking a mapping instead of another will be placed internally in the mapping definition rather than in the mapping invocation. This is one of the facilities that make an operational specification appear almost "declarative".

In the example above the 'class2table' mapping is invoked for each owned element of a package (could be of different actual type: Classes, Associations, (sub)Packages and so on) but, thanks to the '**map**' invocation mode, the body will be executed only for "persistent" instances of the Class metaclass (since guard is the sum of the constraint on types in the signature and the when clause).

```
mapping Package::packageToSchema() : result:Schema
{
    name := self.name;
    table := self.ownedElement->map class2table();
}
mapping Class::class2table(): Table
  when {self.isPersistent()}
{ … }
```

3.3 Mapping merge and mapping inheritance

A mapping body is decomposed in three sub-sections: an initialization section, a population section and a finalization section. This structure allows making implicit a list of interesting things:

- The actual instantiation of the declared output, which is performed within a hidden "instantiation" section that occurs after the initialization.

- The generation of the trace data linking the source element and the output element. The trace data is implicitly used by the *resolve* operators, which allow obtaining from a source instance the objects already created by mappings with this source.

The fact that the actual instantiation of the output element is implicit also permits introducing the concept of rule inheritance. A mapping may declare inheritance from an existing mapping, meaning internally that the inherited mapping is invoked after the instantiation section and before entering the population section. This "declarative" inheritance declaration makes the specification more compact and "structured". A composite transformation that is build as a specialization of an existing set of mapping rules will typically contain new mappings inheriting from the existing ones.

The example below (taken from the specification) shows a simple usage of mapping inheritance. Note that the context type of the inheriting mapping need to conform with the context type of the inherited mapping.

```
mapping Attribute::attr2Column (in prefix:String) : Column {
    name := prefix+self.name;
    kind := self.kind;
    type := if self.attr.type.name='int'
            then 'NUMBER' else 'VARCHAR' endif;
```

```
}
mapping Attribute::attr2ForeignColumn (in prefix:String) : Column
inherits Attr2Column {
    kind := "foreign";
}
```

The concept of mapping merge is the dual feature of mapping inheritance: the merged mapping is invoked after the end of the finalization section. The merge facility allows writing modular mappings specially when there are various outputs involved in a rule: each "part" of a mapping will be treated separately in a merged mapping. Use of this facility may improve significantly the readability of a complex mapping operating on a single metaclass.

We should note that the *guard* mechanism has an influence in the invocation of inherited and merged mappings since only the mappings which guard is satisfied are effectively implicitly invoked.

3.4 Mapping disjunction

Another composition-oriented facility in QVT-Operational is the ability to define a mapping as a disjunction of mappings. Invoking such mapping results on the selection of the first mapping whose guard (type and when clause) succeeds. The *null* value is returned if no guard succeeds. This is an example taken from the QVT specification.:

```
mapping UML::Feature::convertFeature () : JAVA::Element
    disjuncts convertAttribute, convertOperation,
              convertConstructor() {}
mapping UML::Attribute::convertAttribute : JAVA::Field {
    name := self.name;
}
mapping UML::Operation::convertConstructor : JAVA::Constructor
when {self.name = self.namespace.name;}  {
     name := self.name;
}
mapping UML::Operation::convertOperation : JAVA::Constructor
when {self.name <> self.namespace.name;}  {
     name := self.name;
}
```

Mapping inheritance, mapping merge and mapping disjunction are facilities that can be used to modularize a large specification into small pieces.

3.5 Fine grained black-boxes

In Section 1 we mentioned the possibility for defining entire transformations as black-boxes. But black-boxes can also be defined at the level of queries and mapping operations: in such case the operation should not define a body. Black-box mappings allow composing external transformations that operate on model elements directly rather than models. Black-box queries allow integrating specific computations written in an external language which can be needed as a preliminary step to define the logic of a transformation. It is indeed up to the QVT tool implementer to provide the

mechanisms for linking the QVT code with the external code to build the executable transformation program.

4 Conclusion

This paper describes the mechanisms that are available within QVT-Operational formalism to compose transformations. For coarse-grained transformations the level of expressivity offered is basically the one provided by existing scripting 'shell' languages running on top of operating systems. A specific facility is provided for parallelizing transformation invocations. In the other hand, for fine-grained transformations, in addition to the "regular" object-oriented mechanisms, QVT provides a list of innovative facilities to enhance the capacity to modularize a complex specification and to reuse mapping rules.
As we have seen, thanks to the black-box mechanism, QVT also addresses the problem of interoperability with external transformations written in other languages than QVT.
We do not have yet enough experience on using some of these more "advanced" features like disjunction and merge of rules to state how these features will function together and what would be the end-user feedback. Complete implementation of QVT will give us a response on this. However, our past and present experience on using the QVT-Operational ancestor formalism – TRL[3] – and using earlier implementations of QVT allows us to state that the formalism facilitates significantly the writing of complex transformations, which would be more difficult to develop and maintain if implemented using a general purpose language.

References

1. OMG, MOF 2.0 Query View and Transformation specifications version 1.0, November 2005, OMG document ptc/05-11-01 available from www.omg.org
2. OMG, OCL 2.0 Object Constraint Language specifications version 2.0, May 2006, OMG document formal/06-05-01 available from www.omg.org
3. Mariano Belaunde, Mikael peltier: From EDOC to CCM components: A Precise mapping Specification, FASE 2002, Springer

Octel, a Template Language for Generating Structures instead of Textstreams

Jos Warmer

Ordina, The Netherlands

Abstract. This paper introduces Octel, a template langauge for generating models. Octel has the ease of use of text based template languages. At the same time it gives the possibility of composing transformations from smaller ones. Octel has been extensively used in the model transformations in the open source Octopus tool.

1 Introduction

Within the field of model driven development the differences between model-to-model and model-to-text transformations are already acknowlegded. A model-to-model transformation takes some structure, which is often a graph, and transforms it to another structure, whereas a model-to-text transformation takes a structure and produces a textstream, i.e. a sequence of characters without any other internal structuring. In [1] a taxonomy of model transformations is given that clearly distinguishes these types of transformations.

This paper describes the use of a template language that enables model-to-model transformations. Furthermore, this template language is very useful for building larger transformations from small interconnected transformations.

2 Template Language for Model-to-Model Transformations

The use of template languages for model-to-text transformations has become popular. Eclipse based languages like JET, or VisualStudio languages like T4 are good examples. Their popularity can easily be explained by the simplicity of their use. The output file to be generated is simply written as it should be, and at certain places pieces of information from the input model are substituted. However, these template languages are not very suitable for model-to-model transformations. The template is always a text, and therefore the output produced by such a template language is also text.

In our work on the Octopus tool [2] we have created a template language that is able to generate structures instead of text. This language is called Octel, which stands for Octopus Template Language. A difference between Octel and other template languages is that an Octel template is not an example of the final output of the transformation, instead it is an example of the code that is to generate the output.

As example we use the UML to Java transformation that we have implemented in Octopus. For this we have created a Java version of the UML metamodel and a metamodel of Java itself, again in Java. The Octel templates produce the code that generates

the instances of the Java metamodel. Parts of these templates are references to (parts of) the instance of the UML metamodel to be transformed.

For instance, the following code (Code Example 1) generates a setter method for an UML attribute without using Octel. (For space reasons we have left out the generation of the field and getter method.)

```
 1 public void generateAttributeInClass(
 2                   IStructuralFeature att,
 3                   OJClass owner){
 4       FEATURE = new StructuralFeatureMap(att);
 5       owner.addToImports(FEATURE.javaTypePath());
 6
 7       OJOperation method1 = new OJOperation();
 8       owner.addToOperations(method1);
 9       method1.setName(FEATURE.setter());
10       method1.setStatic(FEATURE.isStatic());
11       method1.setVisibility(FEATURE.visibility());
12       method1.setComment("implements the setter for feature '"
13                           + att.getSignature() + "'");
14       OJParameter param1 = new OJParameter();
15       method1.addToParameters(param1);
16       param1.setType(FEATURE.javaTypePath());
17       param1.setName("element");
18       OJBlock body2 = new OJBlock();
19       method1.setBody(body2);
20       OJIfStatement if1 = new OJIfStatement();
21       if1.setCondition(FEATURE.javaFieldName() + " != element");
22       body2.addToStatements(if1);
23       OJBlock then1 = new OJBlock();
24       if1.setThenPart(then1);
25       OJSimpleStatement exp2 =
26             new OJSimpleStatement(FEATURE.javaFieldName()
27                                   + " = element");
28       then1.addToStatements( exp2 );
29 }
```

The type *IStructuralFeature* represents an attribute or association end in the UML model, the type *OJClass* represents a class in the generated Java model, the type *OJOperation* represents a Java method. *StructuralFeatureMap* is explained in section 3.1. Lines 18 to 28 generate the body of the setter method. This body itself is an instance of the type *OJBlock*, which holds an instance of *OJIfStatement* which in turn holds an instance of *OJSimpleStatement* representing the then part of the if-statement in the setter's body.

Certainly the transformation does not function correctly if not all of the details – name, type, visibility, etc.– are included, but writing this code is very tedious work and often you need many lines of code for the transformation of a single, rather simple input element. Furthermore, the code does not look like the intended output of the transformation, which would the following code (Code Example 2) for the input attribute *'+ number : Integer'* of class *Breakfast*.

```
public class Breakfast {
   private int f_number = 0;

   /** Implements the setter for feature '+ number : Integer'
```

```
     *
     * @param element
     */
    public void setNumber(int element) {
          if ( f_number != element ) {
                f_number = element;
          }
    }

}
```

As one can see, it is hard to see what code is generated from the generating Java code, which is oine of the main advantages of template languages. Therefore we decided to write the generating code differently, more in line with the final output to be generated but still generating a structure of Java metamodel instances. The Octel preprocessor then generates the code in Code Example 1, which in its turn generates the code in Code Example 2. The following (Code Example 3) is the input to the Octel preprocessor.

```
 1 public void generateAttributeInClass(
 2                   IStructuralFeature att,
 3                   OJClass owner){
 4       FEATURE = new StructuralFeatureMap(att);
 5       owner.addToImports(FEATURE.javaTypePath());
 6
 7       /**<octel var="owner">
 8             <method type="%FEATURE.javaTypePath()%"
 9                   name="%FEATURE.setter()%"
10                   static="%FEATURE.isStatic()%"
11                   visibility="%FEATURE.visibility()%">
12                   <comment> implements the setter for feature
13                         '%att.getSignature()%' </comment>
14                   <param type="%FEATURE.javaTypePath()%"
15                         name="element"/>
16                   <body>
17                         <if> %FEATURE.javaFieldName()% != element
18                         <then>
19                               %FEATURE.javaFieldName()% = element;
20                         </then></if>
21                   </body>
22             </method>
23       **/
24 }
```

The Octel preprocessor only changes the parts in its input that are between the ‘/**’ and ‘**/’ markers, which is pure XML. The text between ‘%’ signs, which usually refers to information from the input model, is also considered to be Java code. Therefore, the input to the Octel preprocessor is correct Java, and tools available for Java, like editors with code completion, can still be used. Furthermore, the way Octel generates code from the XML tags may be changed by the Octel user. In this way many different target metamodels can be addressed. For instance, we also used Octel to generate Eclipse plug-in manifest files, and Microsoft DSL models.

3 Composition in the Small

From the previous section it is clear that Octel generates code that in turn generates the output of the transformation. This two-step approach is very usefull when the complete transformation (in our example from a UML model to a Java model) is build up from smaller transformations. We call this transformation composition in the small.

Transformation composition in the small is an excellent way to give the user control over the overall transformation. Almost every part of the overall transformation can be turned on or off separately. Furthermore, these parts may be changed or even completely replaced by a different implementation. For instance, in Octopus the following order of small transformations is used to transform the complete UML model into a Java model.

1. Create the sceleton java model consisting of empty, unconnected classes.
2. Add supertype relations.
3. Add implemented interface relations.
4. Add operations.
5. Add attributes.
6. Add association ends.
7. Add code for association classes.
8. Add states.
9. Add code from implemented interfaces.
10. Add code for checking for OCL expressions.
11. Add code for checking multiplicities.
12. Generate the storage layer.
13. Generate the user interface layer.

Step 1 creates the initial model, step 2 to 11 add elements to this initial model, while step 12 and 13 generate classes that use the classes in the initial model. Step 13 inself is, once again, composed of many small transformations. After the complete Java model is created, a fairly simple process produces the textual representation of this Java model in the form of a set of Java class files. In fact, this process implements a model-to-text transformation. Of course, it does not make much sense to leave out some of the steps in this process, for instance to generate code for operations but not for attributes, but in other cases this is very convenient. The Octopus user can freely turn steps 10 to 13 on or off.

3.1 Mappers

Some of the smaller transformations are dependent upon each other. For instance, when generating the body of an operation one needs to know how to address references to attributes or associations ends, in other word, one needs to know how attributes or association ends are transformed. Yet we do not want to mingle the separate steps. For this reason we have created a set of *mapper* classes. A mapper object is a wrapper for an element of the input model that is able to answer questions on how this element is or will be transformed.

For instance, *StructuralFeatureMap* in Code Example 1 is a wrapper for the input attribute. It returns the Java counterpart of an aspect of that attribute. For instance, UML allows spaces in the attribute names, but Java does not, so the wrapper changes the UML name to a similar name that takes into account the Java restrictions on names.

An advantage of using mappers is that changes of coding rules are easily implemented. For instance, the name of the setter method for an attribute is currently always the name of the attribute, starting with a captical letter, prefixed by 'set'.

4 Conclusion

It is only because Octel is able to generate a structure, namely a connected set of instances of the classes in our Java metamodel, that we are able to use the composition in the small approach. If instead we had created a textual template, we were obliged to decide up front whether or not we wanted to include a certain step, like generating code for checking OCL expressions or multiplicities.

Another advantage of composition in the small is that we have separated doing the 'smart' bits of the transformation, namely transforming the UMl model into the Java model, from the 'dumb' bits of the transformation, namely producing the Java class files. This is a separation of concerns, which is very positive when building a complex and large transformation.

References

[1] Anneke Kleppe. MCC: A model transformation environment. In A. Rensink and J. Warmer, editors, *Proceedings of the second European Conference on MDA, 2006*, volume 4066 of *LNCS*, pages 173–187, Berlin Heidelberg, July 2006. Springer-Verlag.

[2] octopus.sourceforge.net