A Taxonomy of Model Transformations

Tom Mens[1], Krzysztof Czarnecki[2], and Pieter Van Gorp[3]

[1] Software Engineering Lab, Université de Mons-Hainaut
Av. du champ de Mars 6, 7000 Mons, Belgium
tom.mens@umh.ac.be
[2] Dept. Electrical and Computer Engineering, University of Waterloo
200 University Ave., West Waterloo ON N2L 3G1, Canada
czarnecki@acm.org
[3] Dept. Mathematics and Computer Science, Universiteit Antwerpen
Middelheimlaan 1, 2020 Antwerpen, Belgium
pieter.vangorp@ua.ac.be

Abstract. This report summarises the results of the discussions of a working group on model transformation of the Dagstuhl Seminar on Language Engineering for Model-Driven Software Development. The main contribution is a taxonomy of model transformation. This taxonomy can be used to help developers in deciding which model transformation approach is best suited to deal with a particular problem.

1 Introduction

This report summarises the results of the discussions of the second working group of the Dagstuhl Seminar on Language Engineering for Model-Driven Software Development. The working group was composed of the following members (in alphabetical order): Jean Bézivin, Alexey Cherchago, Krzysztof Czarnecki, Tracy Gardner, Tudor Girba, Martin Gogolla, Jean-Marc Jézequel, Frédéric Jouault, Alexander Königs, Jochen Küster, Tom Mens, Laurie Tratt, Pieter Van Gorp, Daniel Varro, Heike Wehrheim, and Michel Wermelinger.

The working group addressed a variety of important issues with *model transformation*, undoubtedly the most profound aspect of model-driven software development. The group started with a discussion on the essential characteristics of model transformations, as well as their supporting languages and tools. The group also discussed the commonalities and variabilities between existing model transformation approaches.

Based on this discussion we propose a *taxonomy* of model transformation in this paper. Such a taxonomy is particularly useful to help a software developer choosing a particular model transformation approach that is best suited for his needs.

2 Model transformation discussion and taxonomy

In order to decide which model transformation approach is most appropriate for addressing a particular problem, a number of crucial questions need to be

Dagstuhl Seminar Proceedings 04101
http://drops.dagstuhl.de/opus/volltexte/2005/11

answered. Each of the following subsections investigate a particular question, and suggest a number of objective criteria to be taken into consideration to provide a concrete answer to the question. Based on the answers, the developer can then select the model transformation approach that is most suited for his needs.

2.1 What needs to be transformed into what?

The first important question we tried to tackle was "What needs to be transformed into what?". This question concerns the source and target artifacts of the transformation.

Program and model transformation. A first distinction concerns the kinds of artifacts being transformed. If these artifacts are programs (i.e., source code, bytecode, or machine code), one uses the term *program transformation*. If the software artifacts are models, we use the term *model transformation*. In our view, the latter term encompasses the former one since a model can range from abstract analysis models, over more concrete design models, to very concrete models of source code. Hence, model transformations also include transformations from a more abstract to a more concrete model (e.g., from design to code) and vice versa (e.g., in a reverse engineering context). Model transformations are obviously needed in common tools such as code generators and parsers.

Kleppe et al. [1] provide the following definition of model transformation. *A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition. A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.* The participants of the working group suggest that this should be generalised, in that a model transformation should also be possible with **multiple source models** and/or **multiple target models**. An example of the former is model merging, where we want to combine or merge multiple source models that have been developed in parallel into one resulting target model. An example of the latter is a transformation that takes a platform-independent model, and transforms it into a number of platform-specific models.

Endogenous versus exogenous transformations. In order to transform models, these models need to be expressed in some modeling language (e.g., UML for design models, and programming languages for source code models). The syntax and semantics of the modeling language itself is expressed by a *metamodel* (e.g., the UML metamodel).

Based on the language in which the source and target models of a transformation are expressed, a distinction can be made between *endogenous* and

exogenous transformations. Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages.[4]

This distinction is essentially the same as the one that is proposed in the "Taxonomy of Program Transformation" [2], but ported to a model transformation setting. In this taxonomy, the term *rephrasing* is used for an endogenous transformation, while the term *translation* is used for an exogenous transformation.
Typical examples of translation (i.e., exogenous transformation) are:

- *Synthesis* of a higher-level, more abstract, specification (e.g., an analysis or design model) into a lower-level, more concrete, one (e.g, a model of a Java program). A typical example of synthesis is *code generation*, where the source code is translated into bytecode (that runs on a virtual machine) or executable code, or where the design models are translated into source code.
- *Reverse engineering* is the inverse of synthesis and extracts a higher-level specification from a lower-level one.
- *Migration* from a program written in one language to another, but keeping the same level of abstraction.

Typical examples of rephrasing (i.e., endogenous transformation) are:

- *Optimization*, a transformation aimed to improve certain operational qualities (e.g., performance), while preserving the semantics of the software.
- *Refactoring*, a change to the internal structure of software to improve certain software quality characteristics (such as understandability, modifiability, reusability, modularity, adaptability) without changing its observable behaviour [3].
- *Simplification* and *normalization*, used to decrease the syntactic complexity, e.g., by translating syntactic sugar into more primitive language constructs.

Horizontal versus vertical transformations. A *horizontal transformation* is a transformation where the source and target models reside at the same abstraction level. A typical example is *refactoring*.

A *vertical transformation* is a transformation where the source and target models reside at different abstraction levels. A typical example is *refinement*, where a specification is gradually refined into a full-fledged implementation, by means of successive refinement steps that add more concrete details [4, 5].

Table 1 illustrates that the dimensions *horizontal versus vertical* and *endogenous versus exogenous* are truly orthogonal, by giving a concrete example of all possible combinations. As a clarification for the *Formal refinement* mentioned in the table, a specification in first-order predicate logic or set theory can be gradually refined such that the end result uses exactly the same language as the original specification (e.g., by adding more axioms).

[4] If we have to deal with transformations with multiple source models and/or multiple target models, there can even be more than 2 different languages involved.

Table 1. Orthogonal dimensions of model transformations

	horizontal	vertical
endogenous	*Refactoring*	*Formal refinement*
exogenous	*Language migration*	*Code generation*

Technological space. A second distinction was made based on whether the source and target models of the model transformation belong to one and the same or to different technological spaces [6]. In the latter case, transformation tools need to provide exporters and importers to bridge the technological spaces while the actual transformation is executed in the technological space of either the source or target model.

For example, when translating XML documents into UML diagrams one can choose to execute the actual transformation in either the XML or the MDA technological space.[5] To perform the transformation in the XML technological space, one would use an XSLT or XQuery program translating the general XML document into an XML document conforming to the syntax of the XMI standard (XML metadata interchange) and conforming to the semantics MOF-XMI document for the UML standard. An XMI parser can then be used to import the resulting XMI document in a UML CASE tool, residing in the MDA technological space.

Performing the transformation in the MDA technological space would require a MOF metamodel for XML. After parsing the XML document into instances of this metamodel, the actual transformation could be performed as a MOF transformation. The QVT RFP [7] aims to standardize a language for writing programs performing this kind of model transformations.

2.2 What are the important characteristics of a model transformation?

Level of automation. A distinction can and should be made between model transformations that can be automated and transformations that need to be performed manually (or at least need a certain amount of manual intervention).

An example of the latter is a transformation from a requirements document to an analysis model. For such a transformation, manual intervention is needed to address and resolve ambiguity, incompleteness and inconsistency in the requirements that are (partially) expressed in natural language.

Complexity of the transformation. Some transformations, such as model refactorings, can be considered as small, while others are considerably more heavy-duty. Examples of the latter are parsers, compilers and code generators. The difference in complexity between small transformations and heavy-duty

[5] The MDA technological space encompasses MOF and UML.

transformations is so big that they probably require an entirely different set of techniques and tools.

Preservation. Although there is a wide range of different types of transformations that are useful during model-driven development, each transformation preserves certain aspects of the source model in the transformed target model. The properties that are preserved can differ significantly depending on the type of transformation. For example, with *refactorings* or *restructurings*, the (external) behaviour needs to be preserved, while the structure is modified. With *refinements*, the program correctness needs to be preserved [8].

2.3 What are the success criteria for a transformation language or tool?

In the previous discussion, we restricted ourselves to characteristics of the model transformation or of the models being transformed. Equally important, or perhaps even more important, are the characteristics of a *transformation language* or *transformation tool*. Below we enumerate a number of important *functional requirements* that contribute to the success of such a language or tool.

Ability to create/read/update/delete transformations (CRUD). While this is a trivial requirement for a transformation language, it is not that obvious for a transformation tool. For example, if we consider a typical program refactoring tool, it comes with a predefined set of refactoring transformations that can be applied, but there is often no way to define new refactoring transformations, or to fine-tune existing transformations to specific needs of the user. As such, having the possibility to create new transformations or update existing ones is an important criterion to assess the "openness" or "extensibility" of a dedicated transformation tool.

Ability to suggest when to apply transformations. For certain application scenarios, dedicated tools can be build that suggest to the user which model transformations might be appropriate in a given context. For example, a refactoring tool might not only apply refactoring transformations, but also suggest in which context a particular refactoring should be applied [9, 10].

Ability to customise or reuse transformations. For example, if we adopt an object-oriented transformation language, we may be able to use the inheritance mechanism for this purpose.

Ability to guarantee correctness of the transformations. The simplest notion of correctness is *syntactic correctness*: given a well-formed source model, can we guarantee that the target model produced by the transformation is well-formed? A significantly more complex notion is *semantic correctness*: does the produced target model have the expected semantic properties?

Ability to deal with incomplete or inconsistent models. It is important to be able to transform models early in the software development life-cycle, when requirements may not yet be fully understood or described in natural language. This often gives rise to ambiguous, incomplete or inconsistent models, which implies that we need to have mechanisms for inconsistency management. These mechanisms may be used to detect inconsistencies in the transformations themselves, or in the models being transformed.

Ability to group, compose and decompose transformations. The ability to combine existing transformations into new composite ones is useful to increase the readability, modularity and maintainability of a transformation language. As an example, if we use the mechanism of graph transformation to express model transformations, we may use the notion of *transformation units* for this purpose [11].

Note that decomposition of a complex transformation into smaller steps may also require a control mechanism to specify how these smaller transformations need to be combined. This control mechanism may be implicit or explicit.

Ability to test, validate and verify transformations. Since transformations can be considered as a special kind of programs (e.g., the XSLT transformation language is a Turing-equivalent programming language), we need to apply verification and validation techniques to transformations as well to ensure that they have the desired behaviour. This is for example a crucial requirement for refactoring transformations, were we want to be able to ensure that these transformations preserve the behaviour.

Ability to specify generic and higher-order transformations. Ideally, transformations should be first class entities in a transformation language. If we can represent transformations as models too, we can apply transformations to these models, thus achieving a notion of higher-order transformations. A concrete example of this would be to refactor a given set of transformations (e.g., a family of code generators), to reduce the amount of code duplication in these transformations. In order to achieve this, we need to transform the transformations themselves.

Ability to specify bidirectional transformations. Languages or tools that have this property require fewer transformation rules, since each transformation can be used in two different directions: to transform the source model(s) into target model(s), and the inverse transformation to transform the target model(s) into source model(s).

Support for traceability and change propagation. To support *traceability*, the transformation language or tool needs to provide mechanisms to maintain

an explicit link between the source and target models of a model transformation. To support *change propagation*, the transformation language or tool may have an incremental update mechanism and a consistency checking mechanism.
Note that some transformation approaches require to translate the source model first into some standardised format (e.g., XML), then apply the transformation, and then do another translation to obtain the target model. A clear disadvantage of such an approach is that it is difficult to synchronise source and target models when changes are made to them.

2.4 What are the quality requirements for a transformation language or tool?

Besides all the functional requirements enumerated above, a transformation language or tool should also satisfy a number of *non-functional* or *quality requirements*:

Usability and usefulness. The language or tool should be *useful*, which means that it has to serve a practical purpose. On the other hand, it has to be *usable* too, which means that it should be intuitive and efficient to use. Obviously, this issue is directly related to developer training and experience. Instead of using a full-fledged transformation language, developers may prefer a more direct model manipulation approach, where the internal model is directly accessed by means of an API. The advantage is that developers can keep on using their preferred language and require virtually no extra training. The disadvantage is that the API may restrict the kinds of transformations that can be performed [12].

Verbosity versus conciseness. *Conciseness* means that the transformation language should have as few syntactic constructs as possible. From a practical point of view, however, this often requires more work to specify complex transformations. Hence, the language should be more *verbose* by introducing extra syntactic sugar for frequently used syntactic constructs. It is always a difficult task to find the right balance between these two conflicting goals. Referring to the previous example of direct model manipulation via an API, the developers will typically use a general purpose programming language to specify the transformations. This leads to considerably more verbose code than with dedicated model transformation languages [12].

Scalability. The language or tool should be able to cope with large and complex transformations or transformations of large and complex software models.

Mathematical properties. If the transformation language or tool has a mathematical underpinning, it may be possible, under certain circumstances, to prove theoretical properties of the transformation such as termination, soundness, completeness, (syntactic and semantic) correctness, etc. [8, 13, 14]

Acceptability by user community. The best transformation language from a theoretical point of view may not necessarily be the best from a pragmatic point of view. For example, it the target community is an object-oriented audience, a transformation language based on the logic or functional paradigm may not be the best choice.

Standardization. The transformation tool should be compliant to all relevant standards. For example, it may need to support XMI for importing or exporting the source or target models of a transformation.

2.5 Which mechanisms can be used for model transformation?

Mechanisms should be interpreted here in a broad sense. They include techniques, languages, methods, and so on. To specify and apply a transformation, ideas from any of the major programming paradigms can be used. One can decide to use a procedural, an object-oriented, a functional or a logic approach, or a hybrid approach combining any of the former ones.

The major distinction between transformation mechanisms is whether they rely on a *declarative* or an *operational* (or *imperative*) approach. Declarative approaches focus on the *what* aspect, i.e., they focus on what needs to be transformed into what. Operational approaches focus on the *how* aspect, i.e., they focus on how the transformation itself needs to be performed.

From a theoretical point of view, the *declarative approach* to transformation seems the most promising. It is formally founded, it offers bidirectionality, and more importantly, it offers a simpler semantic model to understand and specify model transformations. For example, the order of rule application is implicit, and the traversal of source models and creation of target models is implicit. This allows one to hide the procedural details of the transformation, making the transformations more compact and maintainable. On the other hand, an *operational approach* may be better suited when considering transformations that are required to incrementally update a model. Thanks to its built-in notions of sequence, selection and iteration, an operational approach also seems better to control the application order of a set of transformations.

Declarative approaches include, but are not limited to, all of the following approaches:

Functional programming. A functional approach towards model transformation seems to be very appealing, since any transformation can be regarded as a function that transforms some input (the source model) into some output (the target model). In most functional languages, functions are first class, implying that transformations can be manipulated as models too. An important disadvantage of the functional approach is that it becomes awkward to maintain state during transformation.

Logic programming. A logic language (e.g., Prolog or Mercury) has many features that are of direct interest for model transformation: backtracking, constraint propagation (in the case of constraint logic programming languages), bidirectionality of rules, and unification. Unification may either be *partial* (which is easier to use and understand) or *full* (which is more powerful). Additionally, logic languages always offer a query mechanism, which means that no separate query language needs to be provided.

Graph transformation. Graph transformation is a set of techniques and associated formalisms that are directly applicable to model transformation [15, 16, 13]. It has many advantages. It is a visual notation: the source, target and the transformation itself can be expressed in a visual way. It is formally founded: one can resort to many theorems to prove certain properties of the transformation. It offers mechanism to compose smaller transformations into more complex ones [11]. The disadvantage is that the various techniques for graph transformations are not necessarily compatible with one another, and that the current tool support is not yet sufficiently mature for industrial use. Another unknown is the scalability of graph transformation approaches.

3 Conclusion

In this paper, we provided a *taxonomy* of model transformation. To goal is to help the developer choosing a particular transformation language, tool, method or technology for his specific needs. This is important, since the application domains of model transformation technology can be very diverse. Therefore, there is no unique answer to the question which approach to model transformation is the best. Nevertheless, we have shown that it is possible to come up with a set of concrete criteria that need to be taken into consideration when dealing with the following crucial questions:

- What needs to be transformed into what?
- What are the important characteristics of a model transformation?
- What are the success criteria for a transformation language or tool?
- Which mechanisms can be used for model transformations?

Based on the answers to these questions, a particular model transformation approach may be selected and adopted to address a particular problem.

References

1. Kleppe, Warmer, J., Bast., W.: MDA Explained, The Model-Driven Architecture: Practice and Promise. Addison Wesley (2003)
2. Visser, E., et al.: Program-transformation.org – a taxonomy of program transformation (2004) URL: http://www.program-transformation.org/Transform/ProgramTransformation.

3. Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley (1999)
4. Wirth, N.: Program development by stepwise refinement. Comm. ACM **14** (1971) 221–227
5. Back, R.J., von Wright, J.: Refinement Calculus. Springer Verlag (1998)
6. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture. (2003)
7. Object Management Group: MOF 2.0 Query / Views / Transformations RFP ad/2002-04-10 (2002) URL: http://www.omg.org/cgi-bin/apps/doc?ad/02-04-10.pdf.
8. Back, R.J.: Correctness preserving program refinements. Technical Report Mathematical Centre Tracts #131, Mathematisch Centrum Amsterdam (1980)
9. van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Proc. 9th Working Conf. Reverse Engineering, IEEE Computer Society Press (2002) 97–107
10. Tom Tourwé, Mens, T.: Identifying refactoring opportunities using logic meta programming. In: Proc. 7th European Conf. Software Maintenance and Re-engineering (CSMR 2003), IEEE Computer Society Press (2003) 91–100
11. Kuske, S.: Transformation Units - A Structuring Principle for Graph Transformation Systems. PhD thesis, University of Bremen (2000)
12. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software (2003) 42–45
13. Mens, T., Demeyer, S., Janssens, D.: Formalising behaviour preserving program transformations. In: Graph Transformation. Volume 2505 of Lecture Notes in Computer Science., Springer-Verlag (2002) 286–301 Proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.
14. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: Viatra - visual automated transformations for formal verification and validation of uml models. In: 17th IEEE International Conference on Automated Software Engineering. (2002) 267–270
15. Courcelle, B.: Graph rewriting: an algebraic and logic approach. Volume B of Handbook of Theoretical Computer Science., Elsevier (1990) 193–242
16. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of MDA. In: Graph Transformation. Volume 2505 of Lecture Notes in Computer Science., Springer-Verlag (2002) 90–105 Proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.