Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science 152 (2006) 125–142
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Taxonomy of Model Transformation

Tom Mens[1]

Software Engineering Lab
Université de Mons-Hainaut
Mons, Belgium

Pieter Van Gorp[2]

Formal Techniques in Software Engineering
Universiteit Antwerpen
Antwerpen, Belgium

Abstract

This article proposes a taxonomy of model transformation, based on the discussions of a working group on model transformation of the Dagstuhl seminar on Language Engineering for Model-Driven Software Development. This taxonomy can be used, among others, to help developers in deciding which model transformation language or tool is best suited to carry out a particular model transformation activity.

Keywords: model transformation, taxonomy, comparison, MDD, MDE

1 Introduction

Model-driven engineering (MDE) is a discipline in software engineering that relies on *models* as first class entities and that aims to develop, maintain and evolve software by performing *model transformations.* MDE is embraced by various organisations and companies, including OMG, IBM and Microsoft. MDE encompasses a wide variety of different techniques, including OMG's Model-Driven Architecture (MDA$^{\mathrm{TM}}$), model-integrated computing, Microsoft's

[1] Email: `tom.mens@umh.ac.be`
[2] Email:`pieter.vangorp@ua.ac.be`

software factories, and many more. There is also a wide variety of tools available, such as the Eclipse Generative Model Transformer (GMT) framework, the Generic Modeling Environment (GME) and many more. For a detailed overview we refer to `www.planetmde.org`

In this paper we propose a *taxonomy* of model transformation. By *taxonomy* we mean "A system for naming and organizing things [...] into groups which share similar qualities" (Cambridge Dictionaries Online). Such a taxonomy can be used for a wide variety of purposes. Among others, it can help a software developer choosing a particular model transformation approach that is best suited for his needs, it can help tool builders to assess the strengths and weaknesses of their tools compared to other tools, and it can help scientists to identify limitations across tools or technology that need to be overcome by improving the underlying techniques and formalisms.

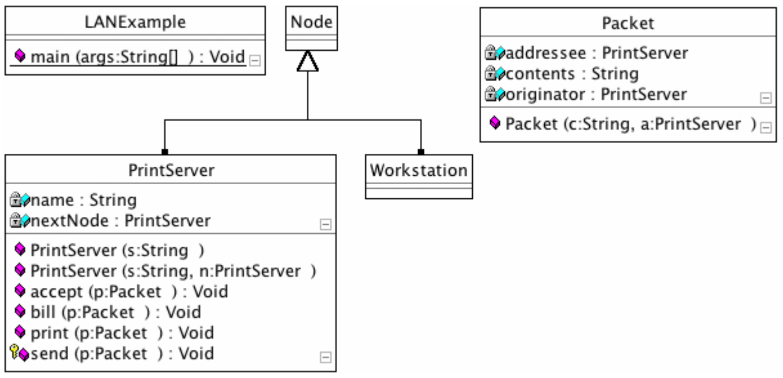
Many of the ideas in the proposed taxonomy our based on the discussions of a working group of a 2004 Dagstuhl seminar on Language Engineering for Model-Driven Software Development. The working group addressed a variety of important issues with *model transformation*, undoubtedly the most profound aspect of model-driven software development [1,2]. 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.

2 Definitions and Examples

Before classifying model transformation techniques, one should understand some model-driven engineering definitions. We will clarify the definition of a model and a model transformation by means of two running examples.

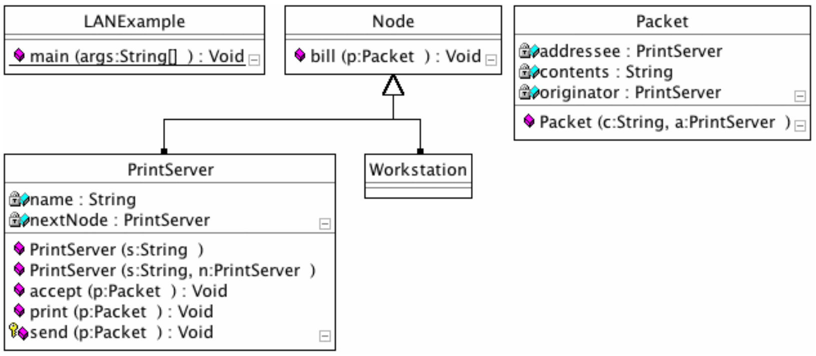
Several sources acknowledge that a model is a simplified representation (or an abstract description) of a part of the world named the system [3,4]. A model is useful if it helps to gain a better understanding of the system. In an engineering context, a model is useful if it helps deciding the appropriate actions that need to be taken to reach and maintain the system's goal.

The goal of software is to automate some tasks in the real world. Models of software requirements, structure and behavior at different levels of abstraction help all stakeholders deciding how this goal should be accomplished and maintained. According to this definition, source code is a model too since it is a simplified representation of the lower-level machine structures and operations that are required to automate the tasks in the real world. Moreover, *correct* source code is a very useful model since it tells the machine what ac-



(a)

Fig. 1. Class diagram before executing the *pull up method* transformation.



(a)

Fig. 2. Class diagram after executing the *pull up method* transformation.

tions need to be taken to maintain the system's goal. Design representations of the source code (e.g., UML diagrams) are useful models if they make the source code more understandable.

When building modeling tools, one needs to model the structure and well-formedness rules of the language in which the models are expressed. Such models are called metamodels [5]. Having a precise metamodel is a prerequisite for performing automated model transformations.

Consider the UML class diagrams in Fig. 1. The diagrams visualize the static structure of a Local Area Network (LAN) application [6] before and after executing a model transformation. The method `bill` is pulled up from the subclass `PrintServer` (see Fig. 1 (a)) to the superclass `Node` (see Fig. 1 (b)).

As another example, consider the problem of translating hierarchical statecharts into flat ones [7]. A part of the translation process consists of redirecting

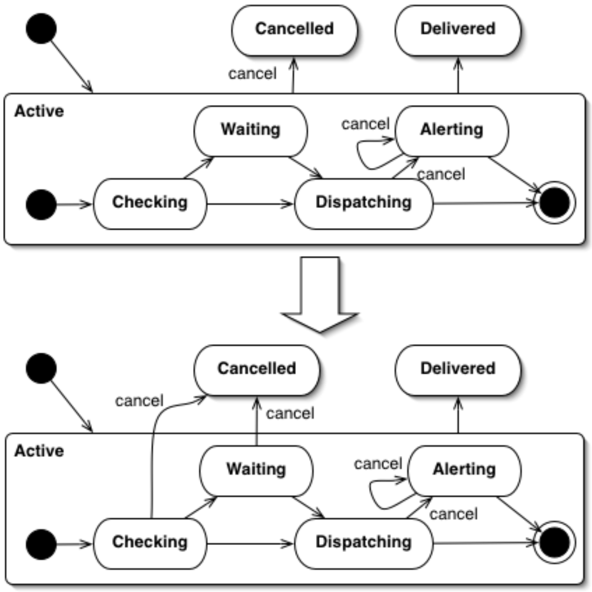


Fig. 3. *A hierarchical statechart being transformed into a more flat one.* The example of a delivery process is inspired by a popular UML book [8]. We have adapted the example such that an order can only be cancelled if it has not been dispatched yet.

transitions starting from composite states to make these transitions start from the contained states. If a contained state already has an outgoing transition with the same label as the outer transition, that contained state will not get an extra outgoing transition. This prevents non-determinism. Fig. 3 visualizes one step of the flattening algorithm. To complete the flattening, the transition to `Delivered` needs to be flattened as well and the `Active` state should ultimately be removed.

3 Model transformation taxonomy

According to Cambridge dictionary, a *taxonomy* is "A system for naming and organizing things [. . .] into groups which share similar qualities". Some taxonomies, such as the taxonomic organisation of species in a biological context, are hierarchical, but this is not a prerequisite.

In this paper, we propose a taxonomy for model transformation that allows us to group tools, techniques or formalisms for model transformation based on their common qualities. In order to identify these qualities, we proceeded as follows. Each of the following subsections investigate an important question with respect to model transformation, and suggest a number of objective criteria to be taken into consideration to provide a concrete answer to the question. Each criterion can be used to group together model transformation approaches satisfying this criterion. As such, or taxonomy provides a multi-dimensional classification, allowing us to group and compare model transformation approaches, based on the criteria of interest.

3.1 What needs to be transformed into what?

The first important question concerns the source and target artifacts of the model transformation. 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*. According to the definitions presented in Section 2, the latter term encompasses the former one since a model can range from abstract analysis representations of the system, 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.

Given that all program transformations can be performed as model transformations, one can clasify the source and target models of a transformation in terms of their structure. More specifically, some systems can be represented as a strict tree whereas others require a graph representation. Note that every graph can be encoded as a tree with references from certain nodes to nodes different from their child nodes. However, navigating a graph encoded as a tree requires (potentially tedious) join operations. Encoding a graph in a relational datastructure leads to even more join operations since the relations between tree nodes and their children need to be represented by means of references as well. Therefore, one should choose the technology that matches the system as closely as possible without sacrificing too much runtime performance.

Number of source and target models.

A first distinguishing characteristic of a model transformation is its number of source and target models. Kleppe et al. [9] provided 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.* We suggest that this should be generalised, in that a model transformation should also be applicable to **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 (PIM), and transforms it into a number of platform-specific models (PSM). Both examples are schematically represented in Fig. 4.

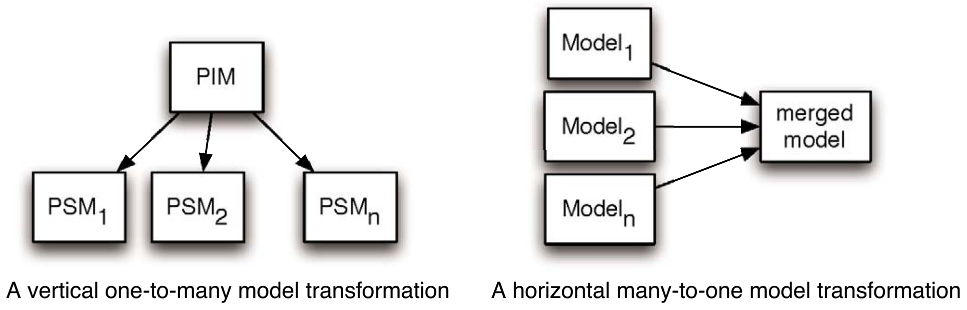

PIM
PSM$_1$
PSM$_2$
PSM$_n$
Model$_1$
Model$_2$
Model$_n$
merged model
A vertical one-to-many model transformation
A horizontal many-to-one model transformation

Fig. 4. Examples of model transformations

Technical space.

A technical space [10] is a model management framework containing concepts, tools, mechanisms, techniques, languages and formalisms associated to a particular technology. A technical space is determined by the meta-metamodel that is used (**M3**-level). For example, the world-wide web consortium (W3C) promotes the *XML technical space*, which uses *XML Schema* as meta-metamodel. This space includes support for languages such as HTML, XML, XMI, XSLT, and XQuery. As another example, the Object Management Group (OMG) promotes the *MDA technical space*, which uses the *MOF* as meta-metamodel, and supports languages such as UML. Many other technical spaces are available, including those relying on abstract syntax trees and grammars, graphs and graph transformations, database technology, or ontologies.

Given a model transformation, its source and target models may belong to the same or to different technical spaces. In the latter case, we need tools and techniques to define transformations that bridge technical spaces. One possibility is to provide model exporters and importers while executing the actual transformation in the technical 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 technical space. To perform the transformation in the XML technical 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 technical space.

Performing the transformation in the MDA technical 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 request for proposals [11] aims to standardize all efforts in trying to implement this kind of model transformations.

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*. For example, the syntax of the UML metamodel is expressed using class diagrams, whereas its semantics is described by a mixture of well-formedness rules (expressed as OCL constraints) and natural language [12].

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. [3]

In [13], essentially the same distinction was proposed, but ported to a model transformation setting. In that taxonomy, the term *rephrasing* was used for an endogenous transformation, whereas the term *translation* was 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.

[3] 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.

- *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 [14]. The *pull up method* transformation of Figure 1 is an example of such a refactoring.
- *Simplification* and *normalization*, used to decrease the syntactic complexity, e.g., by translating syntactic sugar into more primitive language constructs. The statechart flattening transformation of Figure 3 is an example of such a simplification.
- *Component adaptation*, to modify and adapt the code of existing software components, either statically or dynamically (i.e., during component execution), to the user's needs.

One can further classify endogenous model transformations in terms of the number of models involved. If this number is only one, the source and target model are the same and all changes are made *in-place*. Other endogenous transformations create model elements in one model based on properties of another model (regardless of the fact that both models conform to the same metamodel). Such transformations are called *out-place*. Note that exogenous transformations are always out-place. We do not incorporate this distinction in the proposed taxonomy since for most applications it doesn't matter whether a transformation is implemented in- or out-place. Still, the terms have shown to be useful in technical discussions on model transformation.

Horizontal versus vertical transformations.

A *horizontal transformation* is a transformation where the source and target models reside at the same abstraction level. Typical examples are *refactoring* (an endogenous transformation) and *migration* (an exogenous transformation). 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 [15,16].

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).

Table 1
Orthogonal dimensions of model transformations with examples

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

Syntactical versus semantical transformations.

A final distinction can be made between model transformations that merely transform the syntax, and more sophisticated transformations that also take the semantics of the model into account. As an example of *syntactical transformation*, consider a parser that transforms the concrete syntax of a program (resp. model) in some programming (resp. modeling language) into an abstract syntax. The abstract syntax is then used as the internal respresentation of the program (resp. model) on which more complex *semantical transformations* (e.g. refactoring or optimisation) can be applied. Also when we want to import our export our models in a specific format, a syntactical transformation is needed.

3.2 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 transformations is so big that they 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 [17]. The technical space also heavily influences what needs to be preserved. For example, in the case of a database transformation, we need to preserve the integrity of the database, while in the case of a program transformation, we need to preserve the syntactic well-formedness and type correctness of the program.

3.3 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. Many of these criteria are still in a research stage, in the sense that they are not yet fully supported in state-of-the-art model transformation tools.

Creating/Reading/Updating/Deleting 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.

Suggesting when to apply transformations.

For certain application scenarios, dedicated tools can be built 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 [18,19].

Customising or reusing transformations.

For example, if we adopt an object-oriented transformation language, we may be able to use the inheritance mechanism to reuse the specifications of model transformations. Other customisation or reuse mechanisms include parameterisation and templates.

3.3.1 Verifying and guaranteeing correctness of the transformations.

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. 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? Another notion is *syntactic completeness*: for each element in the source model, there should be a corresponding element in the target model that can be created by a model transformation.
A significantly more complex notion is *semantic correctness*: does the produced target model have the expected semantic properties? This is for example a crucial requirement for refactoring transformations, were we want to be able to ensure that these transformations preserve certain behavioural properties. Other important semantic properties are *termination* and *confluence*: given a set of transformations, they should always lead to a result (i.e., they should terminate) and this result should be unique (confluence).

Testing and validating transformations.

Since transformations can be considered as a special kind of programs (e.g., the XSLT transformation language is a Turing-equivalent programming language[4]), we need to apply systematic testing and validation techniques to transformations to ensure that they have the desired behaviour.

[4] See `http://www.unidex.com/turing/utm.htm` for more information

Dealing 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 are 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, and possibly resolve, inconsistencies in the transformations themselves, or in the models being transformed.

Grouping, composing and decomposing transformations.

The ability to compose existing transformations into new composite ones is useful to increase the readability, modularity, maintainability and scalability of a transformation language. 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.

Genericity of 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.

Bidirectionality of transformations.

Languages or tools that have the property of bidirectionality 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).

Supporting 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 a consistency checking mechanism and an incremental update mechanism [20].

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.

3.4 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*. These requirements are of particular interest to industrial users of model transformation tools. They may be a reason for accepting or rejecting the tool in practice.

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 [2].

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 [2].

Performance and scalability.

The language or tool should be able to cope with large and complex transformations or transformations of large and complex software models without sacrificing performance. Another issue that has to do with performance is whether the transformation tool is interpreter-based or compiler-based. Compilation-based approaches may have the benefit of improved performance over interpretation-based ones.

Extensibility.

The flexibility of a tool depends on the ease with which the tool can be extended with new functionality. For example, many contemporary tools (e.g., Eclipse) offer plug-in frameworks in order to facilitate the addition of third-party code (so-called *plug-ins*) into the tool in a standardised way.

Interoperability.

The tool should also be interoperable or easy to integrate with other tools used within the (model-driven) software engineering process. To achieve this, the tool needs to provide bridges to other frequently used technical spaces.

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 a logic or functional paradigm may not be acceptable.

Standardization.

The transformation tool should be compliant to all relevant standards (such as XML, MOF, UML). For example, the tool may need to support XMI for importing or exporting the source or target models of a transformation.

3.5 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 borrow techniques from the a procedural, object-oriented, functional or logic paradigms, or even use a hybrid approach combining any of the former ones.

Nevertheless, it is important to note that a model transformation tool should be dedicated to the construction and modification of models. In this sense, a dedicated domain-specific programming language, preferably even restricted to the technical space of interest, is likely to be more effective than a full-fledged general-purpose programming language.

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 by defining a relation between the source and target models. Operational approaches focus on the *how* aspect, i.e., they focus on how the transformation itself needs to be performed by specifying the steps that are required to derive the target models from the source models.

Declarative approaches (e.g., [21]) are attractive because particular services such as source model traversal, traceability management and automatic bidirectionality can be offered by an underlying reasoning engine. There are several aspects that can be made implicit in a transformation language: (1) navigation of a source model, (2) creation of target model and (3) order of rule execution. As such, declarative transformations tend to be easier to write and understand by software engineers.

Operational (or *constructive*) approaches (e.g., [22]) may be required to implement transformations for which declarative approaches fail to guarantee their services. Especially when the application order of a set of transformations needs to be controlled explicitly, an imperative approach is more appropriate thanks to its built-in notions of sequence, selection and iteration. Such explicit control may be required to implement transformations that reconcile source and target models after they were both heavily manipulated outside that transformation tool.

One particular flavour of a declarative approach is *functional programming.* Such an approach towards model transformation is 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.

Another flavour of a declarative approach is *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), 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.

4 Related work

The taxonomy presented in this paper is similar to, but broader in scope than, the survey of rewriting strategies in program transformation systems by Visser [13]. Indeed, program transformation is only a small subset of model transformation. In this sense, Visser's survey restricts itself to a particular technical space.

Probably the closest work that is related to our taxonomy is the classification of model transformation approaches that has been proposed by Czarnecki and Helsen in [23]. This is not surprising at all, since Czarnecki participated in the Dagstuhl workshop on which many of the ideas in our taxonomy our based. One of the main differences is that Czarnecki and Helsen propose a hierarchical classification based on feature diagrams, while our taxonomy is essentially multi-dimensional. Another important difference is they classifies the specification of model transformations, whereas our taxonomy is more targetted towards tools, techniques and formalisms supporting the activity of model transformation.

5 Conclusion

In this paper, we provided a *taxonomy* of model transformation. The purpose of this taxonomy is manifold: (1) to position concrete model transformation tools and techniques within the domain; (2) to provide a framework for comparing and combining individual tools and techniques; (3) to identify and evaluate tools or technologies for a specific model transformation activity; (4) to provide an overview of the research field of model transformation. Each of these purposes is essential, given the proliferation of tools and techniques within the domain of model transformation, and given the fact that model transformations can be applied for a very wide range of applications.

While there is never a unique answer to the question which approach (or tool, or technique) to model transformation is the best, 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.

In the future, we are planning to apply our model transformation taxonomy in order to find answers to other interesting and important questions such as:

- *Which underlying technique is better suited to support model transformation?* Answering this question will first require us to identify the different techniques that have been used to implement model transformation tools, for example, tree transformation or graph transformation, identify a number of representative tools implementing each technology, and comparing these tools using our taxonomy.
- *Which technical space is better suited for model transformation?* Almost all model transformation tools have a primary technical space, which is typically MOF-based or XML-based. Comparing transformation tools across technical spaces may allow us to identify the virtues and shortcomings of each.

References

[1] 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.

[2] Sendall, S., Kozaczynski, W.: Model Transformation - The Heart and Soul of Model-Driven Software Development. IEEE Software, Special Issue on Model Driven Software Development (2003) 42–45

[3] Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: Proc. 16th Int'l Conf. Automated Software Engineering, IEEE Computer Society (2001) 273

[4] Seidewitz, E.: What models mean. IEEE Software **20** (2003)

[5] Favre, J.M.: Towards a basic theory to model model driven engineering. In: Proc. 3rd Workshop in Software Model Engineering (Satellite workshop at the 7th International Conference on the UML). (2004)

[6] Janssens, D., Demeyer, S., Mens, T.: Case study: Simulation of a LAN. Electronic Notes in Theoretical Computer Science **72** (2003)

[7] Wasowski, A.: Flattening statecharts without explosions. In: Proc. ACM SIGPLAN/SIGBED Conf. Languages, compilers, and tools. (2004) 257–266

[8] Fowler, M., Scott, K.: UML Distilled – Second Edition – A Brief Guide to the Standard Object Modeling Language. Object technology series. Addison-Wesley (1999)

[9] Kleppe, A., Warmer, J., Bast., W.: MDA Explained, The Model-Driven Architecture: Practice and Promise. Addison Wesley (2003)

[10] 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)

[11] 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.

[12] Object Management Group: Unified Modeling Language specification version 1.5. formal/2003-03-01 (2003)

[13] Visser, E.: A survey of rewriting strategies in program transformation systems. Electronic Notes in Theoretical Computer Science **57** (2001)

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

[15] Wirth, N.: Program development by stepwise refinement. Comm. ACM **14** (1971) 221–227

[16] Back, R.J., von Wright, J.: Refinement Calculus. Springer Verlag (1998)

[17] Back, R.: On correct refinement of programs. Journal of Computer and Systems Sciences **23** (1981) 49–68

[18] van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Proc. 9th Working Conf. Reverse Engineering, IEEE Computer Society (2002) 97–107

[19] Tourwé, T., Mens, T.: Identifying refactoring opportunities using logic meta programming. In: Proc. 7th European Conf. Software Maintenance and Re-engineering (CSMR 2003), IEEE Computer Society (2003) 91–100

[20] Van Gorp, P., Janssens, D., Gardner, T.: Write Once, Deploy N: a performance oriented mda case study. In: Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference, IEEE (2004)

[21] Akehurst, D., Kent, S.: A relational approach to defining transformations in a metamodel. In: Proc. 5th Int'l Conf. UML. Volume 2460 of Lecture Notes in Computer Science., Springer-Verlag (2002) 243–258

[22] Sprinkle, J., Agrawal, A., Levendovszky, T., Shi, F., Karsai, G.: Domain model translation using graph transformations. In: Proc. Int'l Conf. Engineering of Computer-Based Systems, IEEE Computer Society (2003) 159–168

[23] Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. (2003)