

Model Transformation: The Heart and Soul of Model-Driven Software Development

Shane Sendall, *Swiss Federal Institute of Technology in Lausanne*

Wojtek Kozaczynski, *Microsoft*

Model-driven approaches move development focus from third-generation programming language code to models, specifically models expressed in the Unified Model Language and its profiles. The objective is to increase productivity and reduce time-to-market by enabling development and using concepts closer to the problem domain at hand, rather than those offered by programming languages. Model-driven development's key challenge is to transform these higher-level

models to platform-specific models that tools can use to generate code.

UML gives numerous options to developers.¹ A UML model can graphically depict a system's structure and behavior from a certain viewpoint and at a certain level of abstraction. This is desirable, because typically we can better manage a complex system description through multiple

models, where each captures a different aspect of the solution. We can use models not only horizontally to describe different system aspects but also vertically, to be refined from higher to lower levels of abstraction. At the lowest level, models use implementation technology concepts.

Working with multiple, interrelated models requires significant effort to ensure their overall consistency. In addition to vertical and horizontal model synchronization, we can significantly reduce the burden of other activities, such as reverse engineering, view generation, application of patterns, or refactoring, through automation. Many of these activities are performed as automated processes that take one or more source models as input and produce one or more target models as output, while fol-

The model-driven approach can increase development productivity and quality by describing important aspects of a solution with human-friendly abstractions and by generating common application fragments with templates. In this article, the authors briefly examine different approaches to model transformations and offer recommendations on the desirable characteristics of a language for describing them.

lowing a set of transformation rules. We refer to this process as *model transformation*.

Here, we analyze current approaches to model transformation, concentrating on desirable characteristics of a model transformation language that can be used by modeling and design tools to automate tasks, thus significantly improving development productivity and quality.

Classifying approaches to model transformation

For the model-driven software development vision to become reality, tools must support this automation. Development tools should let users not only apply predefined model transformations but also define their own. Beyond automating transformation execution, tools could suggest which model transformations a user might appropriately apply in a given context, but this aspect is out of the scope of this article.

Performing a model transformation requires a clear understanding of the abstract syntax and semantics of both the source and target. Metamodeling is a common technique for defining the abstract syntax of models and the interrelationships between model elements. For visual modeling languages, there are a number of advantages in basing a tool's implementation on the language's metamodel.

Such tools offer users three different architectural approaches for defining transformations:

- *Direct model manipulation* (sometimes referred to as *pull*): access to an internal model representation and the ability to manipulate the representation using a set of procedural APIs
- *Intermediate representation*: exporting of the model in a standard form, typically XML, so an external tool can transform it
- *Transformation language support*: a language that provides a set of constructs for explicitly expressing, composing, and applying transformations

One advantage of the direct-model manipulation approach is that the language used to access and manipulate the exposed APIs is commonly a general-purpose language such as Visual Basic or Java, so the developers need little or no extra training to write transformations. Furthermore, developers are generally more comfortable with encoding complicated (trans-

formation) algorithms in procedural languages. Examples are Rational Rose,² which offers a version of VB with a set of APIs to manipulate models, and Rational XDE,³ which exposes an extensive set of APIs to its model server that can be used from Java, VB, or C#. A disadvantage is that the APIs usually restrict the kind of transformations that can be performed. Also, because the programming languages are general-purpose, they lack suitable high-level abstractions for specifying transformations. Consequently, transformations can be hard to write, comprehend, and maintain. One proposal that promises to raise the level of abstraction of operations on UML models is UML's action language.¹ This special-purpose language has been proposed as a way to procedurally define UML transformations^{4,5} and manipulate UML models. However, the language still suffers, albeit less chronically, from a lack of high-level abstractions for dealing with model transformations—for example, transformation composition.

With respect to the intermediate-representation approach, many UML tools can export and import models to and from XMI, which is an XML-based standard for interchange of UML models.¹ Because a model is externalized into XML, it is possible to use existing XML tools, such as XSLT,⁶ to perform model transformations. Even though XSLT was defined specifically for describing transformations, it is nevertheless tightly coupled to the XML that it manipulates. Consequently, it requires experience and considerable effort to define even simple model transformations in XSLT. Another disadvantage of the approach is that transformations are performed in batch mode, which has two important consequences. First, transformations are hard to perform in an interactive dialogue with the user. Second, the tool still needs to reactively manage the synchronization between models after changes. For example, a long and complex transformation performed outside of the tools might be rejected because of the violation of cross-model integrity constraints.

Transformation language support, as the name suggests, provides a specific language for describing model transformations. It offers the most potential of the three approaches because the language can be tailored for that purpose. In this context, you can use many languages to specify and execute model transformations, some of which offer visual con-

**Performing
a model
transformation
requires
a clear
understanding
of the abstract
syntax and
semantics of
both the source
and target.**

**A
transformation
language must
provide for
complete
automation and
must be
expressive,
unambiguous,
and Turing
complete.**

structs. These languages are either declarative, procedural, or a combination of both.

For example, Dragan Milicev proposes a graphical language for describing model transformations that is principally procedural in nature but also offers some declarative features.⁷ A tool that generates C++ code from the specification supports the approach. One limitation is its underlying assumption that you can easily express your choice of source model elements for the transformation in a general-purpose programming language, that is, C++.

The Rational XDE's pattern mechanism is a commercial example of a specialized transformation language.³ This mechanism is built on top of XDE's model server API (discussed earlier), so XDE supports both the direct model manipulation and transformation language support classifications. XDE transformations are defined as model templates called *patterns*, which could contain parameters and arbitrary procedural code written in Java, VB, or C#. You can invoke patterns using a set of predefined callbacks; this effectively means you can make arbitrary "manual" model changes. The key drawback of the XDE's pattern engine is that it provides limited capability to compose patterns.

Another general approach is to treat UML models as graphs.^{8–10} Applying graph rewriting rules helps identify graph transformations. A rule consists of a graph to match, commonly referred to as LHS (left-hand side), and a replacement graph, commonly referred to as RHS (right-hand side). If a match is found for the LHS graph, then the rule is fired. Consequently, the RHS graph replaces the matched subgraph of the graph under transformation. Jon Whittle has also proposed the use of rewriting rules for UML model transformation in the context of logic languages.¹¹

Desirable characteristics of model transformation languages

The languages surveyed vary from principally visual notations to text-only notations, from highly declarative to fully imperative, and from a small set of general language constructs to a large number of specialized language constructs. Which of these characteristics are desirable for model transformation languages?

A declarative language offers implicit interpretation, so you can take advantage of a set of underlying mechanisms to formulate the desired

specification. For example, in the graph transformation approach, the algorithm for LHS graph matching is implicit and need not be expressed as part of the specification. On the other hand, an imperative language would make explicit every step of such a matching algorithm.

The key to designing a transformation language is to offer transformation abstractions that are intuitive and cover the largest possible range of situations, and therefore a trade off between the typical conciseness of a declarative language and comprehension.

Many of the rules for mapping source model elements to target model elements can be made implicit and can be defined in a similar manner to the way that people communicate. As such, a declarative language can facilitate this aspect. For example, the intuitive interpretation of a certain schema might imply a depth-first traversal of the specification hierarchy. In this case, making this implicit in the language would be desirable. Nevertheless, in many of the approaches we surveyed, imperative operators are commonly used in transformation composition, because this aspect of transformation description is more suited to an imperative interpretation.

The following would be desirable characteristics for model transformation languages. We take as a given that a transformation language must provide for complete automation and must be expressive, unambiguous, and Turing complete.

Preconditions

A transformation is typically only meaningfully applied against certain model configurations. Thus, it would be desirable in many cases to describe the conditions under which the transformation produces a meaningful result, which can then be enforced by a tool at execution time.

Composition


It is often desirable to combine existing transformations to build new composite ones, because it is almost always easier to compose components than to build something from basic parts. Furthermore, it might be easier to build and test a transformation piecemeal by describing its parts first and then bringing them together to form the whole. So, a transformation language should support transformation composition.

Form

The accessibility and acceptance of a language depends on its form. One of UML's appealing features is that it uses a graphical form. Graphical representations of models have proved popular because there are perceived cognitive gains compared to fully textual representations. In the context of a transformation language, specifying the structure of the input selection and result of a transformation using visual means is an appealing prospect.

Usability

Usability depends on both the language's purpose and the preferences and backgrounds of its users, who might balance ease-of-understanding, precision, concision, and ease-of-modification differently. A transformation language's usability is strongly affected by whether it is declarative or imperative. Declarative approaches make the language more concise, making implicit a number of aspects of the transformation algorithm. Such approaches can greatly simplify the description of transformation rules. Imperative languages, on the other hand, offer a familiar paradigm for composing transformation rules, that is, sequence, selection, and iteration. In this direction, a language that mixes both kinds of approaches could demonstrate the advantages of both worlds. Examples of such a hybrid approach already exist.⁸⁻¹⁰

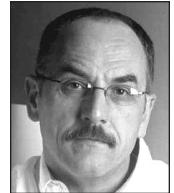
With the potential impact of model-driven approaches on software development practices, tools will need to better automate the construction and evolution of software models. The best way to go about this goal is for tools to offer an executable model transformation language that lets us automate model creation, development, and maintenance activities. The language characteristics we have proposed offer a measuring stick for judging the quality of future model transformation technologies. It is indeed an opportune time to consider which direction the community should be taking for model transformation technologies, especially with the current standardization effort by the Object Management Group and many industrial and academic efforts in this area.^{12,13} 

About the Authors



Wojtek Kozaczynski is a software architect at Microsoft's Platform Architecture Group. His research interests include software architectures, software reuse, software process, and software development automation. He received a PhD in information and decision sciences from the Technical University of Wrocław, Poland. Contact him at One Microsoft Way, Redmond, WA 98052; wojtek@microsoft.com.

Shane Sendall is a senior research and teaching assistant at the Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. His research interests center around improving model-based software development practices. He received a PhD in computer science from the EPFL. He is a member of the IEEE and the ACM. Contact him at the Software Eng. Laboratory, 1015 Lausanne EPFL, Switzerland; sendall@acm.org.



Acknowledgments

Sendall's activities were sponsored by the FIDJI project no. MEN/IST/01/001, Luxembourg. Also, he thanks Olivier Biberstein and the FIDJI team at Luxembourg University of Applied Science.

References

1. *OMG Unified Modeling Language Specification*, ver. 1.5, OMG Unified Modeling Language Revision Task Force, Mar. 2003, www.omg.org/technology/documents/formal/uml.htm.
2. *Rational Rose Family*, IBM/Rational Software Corp., 2003, www.rational.com/products/rose/index.jsp.
3. *Rational XDE*, IBM/Rational Software Corp., 2003, www.rational.com/products/xde/index.jsp.
4. S. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley, 2002.
5. G. Sunyé et al., "Using UML Action Semantics for Executable Modeling and Beyond," *Advanced Information Systems Eng.: 13th Int'l Conf. (CAiSE 01)*, LNCS 2,068, Springer-Verlag, 2001, pp. 433-447.
6. *XSL Transformations (XSLT) Version 1.0*, W3C, www.w3.org/TR/xslt, 2003.
7. D. Milicev, "Domain Mapping Using Extended UML Object Diagrams," *IEEE Software*, vol. 19, no. 2, Mar./Apr. 2002, pp. 90-97.
8. A. Agrawal, G. Karsai, and F. Shi, "A UML-Based Graph Transformation Approach for Implementing Domain-Specific Model Transformations," to be published in *Int'l J. Software and Systems Modeling*, 2003.
9. T. Fischer et al., "Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language," *Theory and Application of Graph Transformation: 6th Int'l Workshop (TAGT 98)*, LNCS 1,764, Springer-Verlag, 1998, pp. 296-309.
10. S. Sendall et al., "Supporting Model-to-Model Transformations: The VMT Approach," *Workshop Model Driven Architecture: Foundations and Applications*, Technical Report TR-CTIT-03-27, Univ. of Twente, 2003.
11. J. Whittle, "Transformations and Software Modeling Language: Automating Transformations in UML," *Proc. UML 2002*, LNCS 2,460, Springer-Verlag, 2002, pp. 227-242.
12. *Model Driven Architecture (MDA)*, OMG Architecture Board ORMSC, 9 July 2001 (draft), www.omg.org/cgi-bin/doc?ormsc/2001-07-01.
13. *MOF 2.0 Query/Views/Transformations RFP*, OMG, 2002, www.sciences.univ-nantes.fr/info/lrsg/Pages_perso/MP/pdf/02-04-10.doc.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.