Transparent Model Transformation: Turning Your Favourite Model Editor into a Transformation Tool

Vlad Acretoaie[1], Harald Störrle[1], and Daniel Strüber[2]

[1] Technical University of Denmark, Kgs. Lyngby, Denmark
{rvac,hsto}@dtu.dk
[2] Philipps-Universität Marburg, Marburg, Germany
strueber@mathematik.uni-marburg.de

Abstract. Current model transformation languages are supported by dedicated editors, often closely coupled to a single execution engine. We introduce Transparent Model Transformation, a paradigm enabling modelers to specify transformations using a familiar tool: their model editor. We also present VMTL, the first transformation language implementing the principles of Transparent Model Transformation: syntax, environment, and execution transparency. VMTL works by weaving a transformation aspect into its host modeling language. We show how our implementation of VMTL turns any model editor into a flexible model transformation tool sharing the model editor's benefits, transparently.

1 Introduction

The science and practice of model transformation (MT) has made significant progress since it was first identified as the *"heart and soul"* of Model-Driven Engineering (MDE) [12]. A varied array of model transformation languages (MTLs) have been proposed since then, each with its own benefits and drawbacks.

While it has found adoption in specialized domains such as embedded systems development, MDE remains outside the mainstream of software development practice. Empirical evidence identifies the poor quality of tool support as one of the main obstacles in the path of large-scale industrial adoption of MDE [18]. Considering the central role of MT in MDE, as well as the experimental nature of most MT tools, we infer that at least some of the criticism addressed to MDE tool quality directly concerns MT tools.

Most (if not all) executable MTLs currently come with dedicated tools that modelers must learn and use in order to specify and execute transformations. But modelers already have at their disposal at least one mature, production-ready tool which they know how to use: their model editor. This observation leads to our central research question:

> **Is it possible to explicitly specify model transformations using only existing, conventional model editors as an interface?**

In this paper we show that this question can be answered positively by following the three principles of Transparent Model Transformation (TMT):

1. The MTL can express transformations at the syntax level supported by the model editor. In most cases this is concrete syntax, but abstract syntax, containment tree, and textual interfaces are also common.
2. Users are free to adopt their preferred editor for each transformation artefact: the source and target model(s), as well as the transformation specification.
3. Transformations can be compiled to multiple executable representations.

We propose the Visual Model Transformation Language (VMTL) as the first MTL following the principles of TMT. Fig. 1 positions VMTL in the current model transformation landscape and highlights its key benefits. Namely, VMTL is a declarative language designed to be woven at the syntactic level into any host modeling language, turning that modeling language into a transformation language for models conforming to it. VMTL adopts any editor of the host modeling language as its own, effectively turning it into a transformation editor. Transformations are subsequently executed by compilation to existing MTLs, which we exemplify in this paper by compiling to the Henshin [3] MTL.

The remainder of this paper is structured as follows: Section 2 introduces VMTL via a motivating example, Section 3 provides an overview of VMTL's main features, Section 4 lays out the fundamentals of TMT with VMTL as an application, Section 5 describes our implementation of VMTL, Section 6 discusses the scope and limitations of VMTL, Section 7 summarizes related work, and Section 8 concludes the paper.

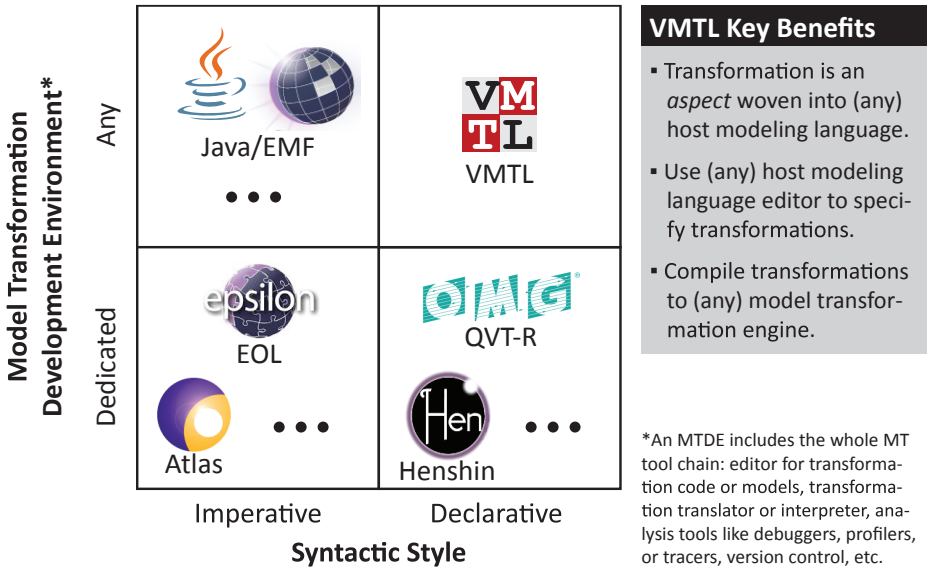

Fig. 1. VMTL and its key benefits in the current model transformation landscape

2 Motivating Example

Consider a UML [10] Use Case model in which an Actor is connected by Associations to two Use Cases, one of which extends the other. The described scenario is a refactoring candidate because the extending Use Case *"typically defines behavior that may not necessarily be meaningful by itself"* [10]. Deleting the Association between the Actor and the extending Use Case is recommended.

A VMTL specification for this transformation is shown in Fig. 2 (top). VMTL employs textual annotations for a number of purposes, such as specifying model manipulation operations. The `delete` annotation is used here to state that the offending Association must be removed from the model. In the case of UML, Comments are an appropriate vehicle for VMTL annotations. Annotation-carrying comments are identified by the `<<VM Annotation>>` stereotype.

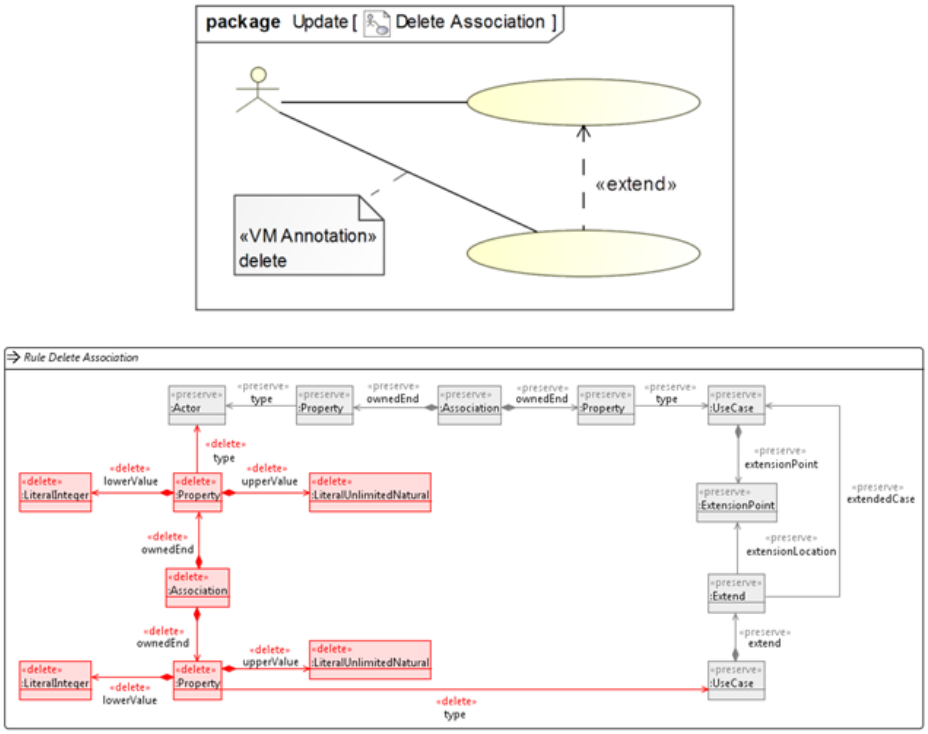

Fig. 2. Example transformation specified using VMTL (top) and Henshin (bottom)

The same transformation could be specified using most existing MTLs, such as Henshin, a graph transformation-based MTL (see Fig. 2, bottom). The Henshin specification is considerably more verbose than its VMTL counterpart, arguably due to the complexity of the UML metamodel. This observation is true

for all MTLs exposing the abstract syntax of the host modeling language, since large and complex metamodels are by no means unique to UML and its profiles.

Nevertheless, specifying transformations at the concrete syntax level is not the main argument put forward by VMTL. A more compelling argument is that VMTL allows specifying transformations directly in the model editor. The transformation in Fig. 2 (top) is specified using the MagicDraw model editor (`http://www.nomagic.com/products/magicdraw`), but any other UML editor could have been used instead, including containment tree and abstract syntax editors. VMTL circumvents the need for a dedicated transformation editor by implementing the principles of Transparent Model Transformation.

3 The Visual Model Transformation Language

VMTL is a usability-oriented MTL descended from the Visual Model Query Language (VMQL [14]). It is a model-to-model, unidirectional transformation language supporting endogenous and exogenous transformations, rule application conditions, rule scheduling, and both in-place and out-place transformations. VMTL transformations can be specified for models expressed in any general-purpose or domain-specific modeling language meeting the preconditions defined in Section 4.1. We refer to these modeling languages as *host languages*.

A VMTL transformation consists of one or more *rules*, each having an *execution priority*. If two rules have equal priorities, the executed rule is selected non-deterministically. A transformation terminates when no rules are applicable. Rules consist of a `Find` pattern, a `Produce` pattern, and optional `Forbid` and `Require` patterns. All patterns are expressed using the host language(s), typically at the concrete syntax level. Model elements and meta-attributes that do not have a concrete syntax representation are also included in the transformation specification. VMTL patterns correspond to the notions of Left-Hand Side (LHS), Right-Hand Side (RHS), Negative Application Condition (NAC), and Positive Application Condition (PAC) from graph transformation theory [5]. Some transformations, such as the one in Fig. 2 (top), allow the `Find` and `Produce` patterns to be merged for conciseness, resulting in an `Update` pattern. Strings starting with the "`$`" character represent variables, and can be used wherever the host language accepts a user-defined string. Variables identify corresponding model elements across patterns and support rule parameterization.

Patterns may contain textual annotations expressed as logic programming-inspired clauses. The adoption of logic clauses as an annotation style is motivated by their declarative nature and their composability via propositional logic operators. VMTL provides clauses for pattern specification, model manipulation, and transformation execution control. Apart from the `delete` clause featured in Fig. 2 (top), examples of VMTL clauses include `create` (for creating model elements), `indirect` (for specifying a relation's transitive closure), `optional` (for identifying model elements that can be omitted from successful pattern matches), and `priority` (for specifying rule priorities). A complete list of VMTL clauses and a more detailed presentation of the language are available in [1].

4 The Principles of Transparent Model Transformation

Transparent Model Transformation is defined by three principles: (1) *syntax transparency*, (2) *environment transparency*, and (3) *execution transparency*. The following subsections define these principles and exemplify them on VMTL.

4.1 Syntax Transparency

Consider an MTL capable of specifying transformations on models conforming to metamodel $\mathcal{M}$. The MTL is said to be *syntax transparent* with respect to $\mathcal{M}$ if all such transformation specifications also conform to $\mathcal{M}$. For example, since VMTL is a syntax transparent language, Fig. 2 (top) simultaneously represents a valid UML model and a transformation specification.

VMTL achieves syntax transparency by *weaving a transformation aspect* into the host modeling language. The constructs of VMTL – rules, patterns, and annotations – are mapped to existing elements of the host language using stereotypes or naming conventions. Consider, for instance, the realization of a VMTL `Update` pattern and a VMTL annotation in UML and Business Process Model and Notation (BPMN [9]). The UML realizations rely on stereotypes (the `<<VM Update>>` stereotype for `Packages` and the `<<VM Annotation>>` stereotype for `Comments`), while the BPMN realizations rely on naming conventions (the `[VM Update]` prefix for `Package` names and the `[VM Annotation]` prefix for `Text Annotation` IDs). We refer to these realizations of VMTL as VMTL_{UML} and $\text{VMTL}_{\text{BPMN}}$, respectively. Similar realizations can be created for other general-purpose or domain-specific modeling languages.

VMTL can only be woven into host modeling languages meeting certain prerequisites. First of all, the host language must support a scoping construct, a role played by Packages in UML and BPMN. Scoping constructs enable VMTL's execution engine to identify which transformation rules or patterns different model elements belong to. Second, all host language elements must support annotations, which are required to act as containers for VMTL clauses. Finally, the availability of a profiling mechanism facilitates the realization of VMTL, since stereotypes can precisely identify model elements as VMTL constructs. A profiling mechanism can be substituted by the adoption of naming conventions.

4.2 Environment Transparency

An MTL is *environment transparent* if it allows users to adopt their preferred editors for interacting with all transformation artefacts: the source model(s), target model(s), and transformation specification. Environment transparency is facilitated by syntax transparency, but can also exist independently. For instance, most textual MTLs allow the use of general-purpose text editors as specification tools, thus exhibiting environment transparency but not syntax transparency.

Since most current MTLs are experimental, few are supported by mature, production-ready editors. The ability to specify transformations using existing model editors is thus beneficial to end-users from two standpoints: (1) avoiding

the learning curve imposed by a new editor, and (2) leveraging a tested, mature tool. By promoting loose editor coupling, environment transparency also opens new deployment avenues, such as remote transformation execution.

VMTL is an environment transparent language. For example, VMTL_{UML} transformations are specified using a UML editor, while a $\text{VMTL}_{\text{BPMN}}$ transformations are specified using a BPMN editor.

4.3 Execution Transparency

An MTL is *execution transparent* if transformations specified using it can be executed by compilation to multiple MTLs operating at a lower abstraction level. Execution transparency gives users the freedom to select a transformation engine appropriate for the task at hand. For instance, in a safety-critical scenario, users might prefer a transformation engine that supports model checking and state-space exploration over one that aims at highly efficient rule execution.

The number and complexity of language constructs included in VMTL is deliberately limited in order to facilitate its compilation to existing MTLs. Since the components of VMTL transformations can be mapped to graph transformation concepts, the most intuitive compilation targets are graph transformation-based MTLs. However, implementations based on imperative MTLs (e.g. EOL [8]), transformation primitive libraries (e.g. T-Core [15]), or general purpose programming languages accompanied by modeling APIs are all possible.

5 Implementation and Deployment

Our implementation of VMTL is based on the Eclipse Modeling Framework (EMF [13]) and the Henshin MT engine. Henshin was selected because its graph transformation-based operational semantics aligns well with VMTL. As a standalone API, it also supports VMTL's syntax and environment transparency. The architecture of our implementation is presented in Fig. 3. As illustrated, the source model and VMTL specification are created using the same editor.

VMTL specifications are compiled by the VM* Runtime[1] into semantically equivalent Henshin specifications to be executed by the Henshin transformation engine. The compilation process can be seen as a Higher-Order Transformation (HOT) consisting of the four steps illustrated in Fig. 3.

In step ❶ model fragments representing transformation components are identified and extracted from the transformation model. These are the transformation's Left-Hand Side (LHS), Right-Hand Side (RHS), Negative Application Conditions (NAC), and Positive Application Conditions (PAC). As these components correspond to VMTL patterns, their identification is informed by VMTL stereotypes or naming conventions.

In step ❷ the extracted model fragments are translated into structurally equivalent Henshin graphs intended to play the same role (LHS, RHS, NAC, or PAC) in the generated Henshin transformation.

[1] The VM* Runtime is also capable of evaluating model queries and constraints.

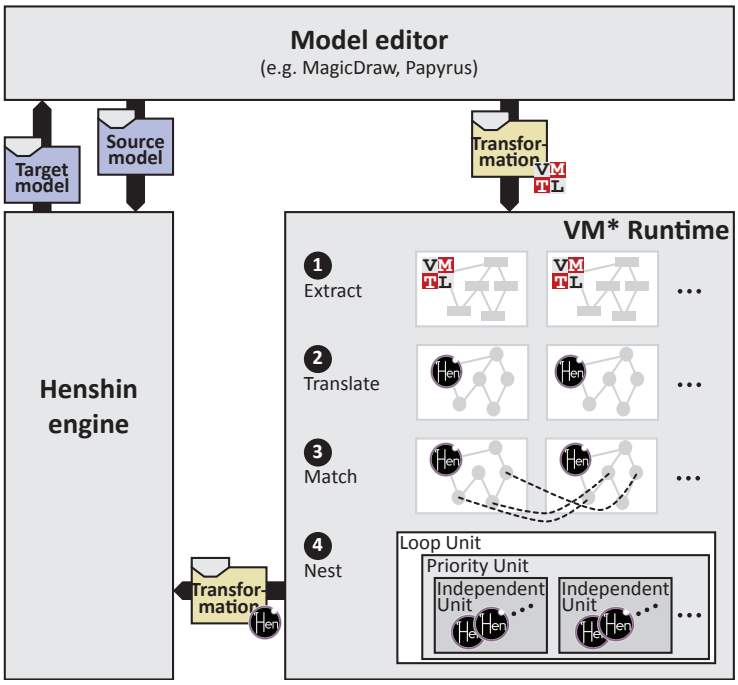

Fig. 3. The architecture of a Henshin-based VMTL implementation. Numbers encircled in black indicate the sequence of steps in the VMTL to Henshin compilation process.

In step ❸ a set of atomic Henshin rules are created by constructing mappings between the nodes of each LHS graph and the corresponding nodes in every other graph belonging to the same rule. As a mapping is a connection between two matching nodes, obtaining the set of mappings between two graphs is equivalent to computing a match between the graphs. The EMFCompare (`https://www.eclipse.org/emf/compare/`) API is used for match computation.

In step ❹ the generated rules are nested inside `Unit`s, Henshin's control flow specification formalism. The resulting control structure implements the operational semantics of VMTL: The applicable rule with highest priority is executed until no more applicable rules exist, at which point the MT terminates.

The architecture presented in Fig. 3 is compatible with several deployment strategies. In a monolithic plugin-based deployment, a model editor plugin encapsulates the VM* Runtime and the MT engine. This approach offers limited portability, as a full-featured new plugin is required for every editor.

To improve portability without sacrificing editor integration, the VM* Runtime and the MT engine can be deployed remotely and accessed via a REST API[2]. Business logic can be removed from the editor plugin, facilitating its re-implementation. However, transferring models over a network is a performance bottleneck, while remote model processing requires strong security provisions.

[2] Any other remote code execution technology may be used.

A third option is to forego editor integration, and develop a separate Web application as a user interface for VMTL. This solution allows specifying VMTL transformations using any editor supporting the host language. The cost is that users must leave the model editor, making interactive transformation execution infeasible. The above-mentioned issues related to remote model processing also apply. We have adopted this deployment strategy for the Hypersonic model analysis API, and provided an in-depth analysis of its advantages and drawbacks [2].

6 Scope and Limitations

Apart from its benefits, the transparent approach to model transformation embraced by VMTL has some inherent limitations, which we discuss in this section.

In VMTL, there are no explicit mappings between the elements of different patterns included in a transformation rule. Instead, the VM* Runtime infers the mappings as described in Section 5. In contrast, most declarative MTLs assume that these mappings are specified by the transformation developer. In the general case, inferring them programmatically requires model elements to have unique identifiers corresponding across patterns. An element's name and type can be used to construct such identifiers, but with no guarantee of uniqueness. Furthermore, some host language elements might not have a name meta-attribute. VMTL therefore allows users to attach tags of the form `#id` to model elements via annotations. It is the developer's responsibility to ensure that corresponding elements have the same name or tag in all patterns. These element identification provisions have the added benefit of allowing the patterns of a rule to conform to different metamodels, thus providing support for exogenous transformations.

One may also argue that VMTL's priority-based rule scheduling is not sufficiently expressive. While not included in the current VMTL specification, control flow structures such as conditional execution and looping constructs could be specified using VMTL's existing textual annotation mechanism.

At the implementation level, incompatibilities between VMTL's operational semantics and the capabilities of its underlying MT engine may appear. One example is the `indirect` clause, allowing VMTL patterns to express a relation's transitive closure, i.e. a chain of undefined length of instances of this relation. Transitive closure computation is problematic for most graph transformation-based engines, but trivial for, say, a logic programming-based engine.

Employing model editors to carry out a task they were not designed for also brings a series of limitations. The well-formedness and syntactical correctness of VMTL rules cannot be verified inside the editor in the absence of a dedicated plugin, while transformation debugging would also benefit from editor extensions. On the other hand, most model editors will enforce the conformance of VMTL patterns to the host language metamodel. This expressiveness limitation is mitigated by VMTL's textual annotations. Finally, displaying target models in the host editor is complicated due to the fact that diagram layout is typically not part of the host language metamodel. Maintaining a layout similar to that of the source model is therefore only possible for in-place transformations.

7 Related Work

MoTMoT [17] proposes an extensible UML 1.5 profile as a uniform concrete syntax for all graph transformation languages. This approach allows graph transformations to be specified using any UML 1.5 editor, and executed by existing graph transformation engines. Although it offers execution transparency and limited environment transparency, MoTMoT does not address syntax transparency.

Several MT approaches (e.g. PICS [4], AToMPM [16]) include concrete syntax model fragments in their specification languages, taking a first step towards syntax transparency. Some of these approaches (e.g. AToMPM) augment the host modeling language with flowchart-like rule scheduling constructs. Even though they are more expressive than VMTL's priority-based scheduling mechanism, these augmentations preclude full syntax and environment transparency. In the same area, Schmidt [11] proposes a transformation profile for UML models, but does not consider other host modeling languages.

Model Transformation By-Example (MTBE, [7]) is an emerging paradigm aimed at leveraging the concrete syntax of host modeling languages. In MTBE, transformations are *inferred* using machine learning or optimization algorithms from a series of example source and target model pairs. In contrast, VMTL transformations are explicitly *specified* using the host language model editor.

Execution transparency is addressed in the context of the systematic development of model transformations by transML [6]. In the same direction, AToMPM transformations are compiled to a lower-level specification language, namely the T-Core [15] transformation primitive library.

8 Conclusion

The perceived lack of adequate tool support in MDE can be mitigated by leveraging production-ready tools familiar to modelers, such as conventional model editors. Adopting existing model editors as transformation tools requires a new approach to model transformation, which we refer to as Transparent Model Transformation (TMT). The principles of syntax transparency, environment transparency, and execution transparency define TMT. Although a number of MTLs adopt subsets of these principles, they have never been explicitly acknowledged and systematized. Doing so has been the first contribution of this paper.

Our second contribution has been the proposal of VMTL: the first transformation language fully compliant with the principles of TMT. We have introduced VMTL's syntax and high-level semantics, and discussed its scope and limitations.

Finally, we have presented the VM* Runtime as an implementation of VMTL. The VM* Runtime leverages the existing Henshin transformation engine, while supporting both local and distributed deployment. It allows us to conclude that TMT is feasible not only conceptually, but also practically.

Acknowledgments. The authors would like to thank Gabriele Taentzer for her insightful comments on the content and presentation of this paper.

References

1. The VM* Wiki, https://vmstar.compute.dtu.dk/
2. Acretoaie, V., Störrle, H.: Hypersonic: Model Analysis and Checking in the Cloud. In: Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering. CEUR Workshop Proceedings, vol. 1206, pp. 6–13 (2014)
3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Berlin Heidelberg (2010)
4. Baar, T., Whittle, J.: On the Usage of Concrete Syntax in Model Transformation Rules. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, pp. 84–97. Springer, Berlin Heidelberg (2007)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Berlin Heidelberg (2006)
6. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: Engineering Model Transformations with transML. Softw. Syst. Model. 12(3), 555–577 (2013)
7. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model Transformation By-Example: A Survey of the First Wave. In: Düsterhöft, A., Klettke, M., Schewe, K.D. (eds.) Conceptual Modelling and Its Theoretical Foundations. LNCS, vol. 7260, pp. 197–215. Springer, Berlin Heidelberg (2012)
8. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Berlin Heidelberg (2006)
9. Object Management Group: Business Process Model and Notation (BPMN), Version 2.0.2 (2013), http://www.omg.org/spec/BPMN/2.0.2/
10. Object Management Group: Unified Modeling Language (UML), Version 2.5 Beta 2 (2013), http://www.omg.org/spec/UML/2.5/Beta2/
11. Schmidt, M.: Transformations of UML 2 Models Using Concrete Syntax Patterns. In: Guelfi, N., Buchs, D. (eds.) RISE 2006. LNCS, vol. 4401, pp. 130–143. Springer, Berlin Heidelberg (2007)
12. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Softw. 20(5), 42–45 (2003)
13. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley Professional, Second edn. (2008)
14. Störrle, H.: VMQL: A Visual Language for Ad-Hoc Model Querying. J. Visual Lang. Comput. 22(1) (2011)
15. Syriani, E., Vangheluwe, H., LaShomb, B.: T-Core: a framework for custom-built model transformation engines. Softw. Syst. Model. 13(3), 1–29 (2013)
16. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Huseyin, E.: AToMPM: A Web-based Modeling Environment. In: Joint Proc. of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition. CEUR Workshop Proceedings, vol. 1115, pp. 21–25 (2013)
17. Van Gorp, P., Keller, A., Janssens, D.: Transformation Language Integration Based on Profiles and Higher Order Transformations. In: Gašević, D., Lömmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 208–226. Springer, Berlin Heidelberg (2009)
18. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 1–17. Springer, Berlin Heidelberg (2013)