CrossMark

# VMTL: a language for end-user model transformation

Vlad Acreţoaie[1] · Harald Störrle[1] · Daniel Strüber[2]

**Abstract** Model transformation is a key enabling technology of Model-Driven Engineering (MDE). Existing model transformation languages are shaped by and for MDE practitioners—a user group with needs and capabilities which are not necessarily characteristic of modelers in general. Consequently, these languages are largely ill-equipped for adoption by end-user modelers in areas such as requirements engineering, business process management, or enterprise architecture. We aim to introduce a model transformation language addressing the skills and requirements of end-user modelers. With this contribution, we hope to broaden the application scope of model transformation and MDE technology in general. We discuss the profile of end-user modelers and propose a set of design guidelines for model transformation languages addressing them. We then introduce Visual Model Transformation Language (VMTL) following these guidelines. VMTL draws on our previous work on the usability-oriented Visual Model Query Language. We implement VMTL using the Henshin model transformation engine, and empirically investigate its learnability via two user experiments and a think-aloud protocol analysis. Our experiments, although conducted on computer science students exhibiting only some of the characteristics of end-user modelers, show that VMTL compares favorably in terms of learnability with two state-of the-art model transformation languages: Epsilon and Henshin. Our think-aloud protocol analysis confirms many of the design decisions adopted for VMTL, while also indicating possible improvements.

## 1 Introduction

Model transformation (MT) is "the heart and soul" [44] of Model-Driven Engineering (MDE), a software development paradigm in which models replace code as the central artifact of the software development process [42]. Model transformation languages (MTLs) play a very specific role in MDE: They bridge the gap between models at different abstraction levels, as well as between models and secondary artifacts such as code and documentation [44]. Reflecting their MDE origins, existing MTLs operate under two assumptions: (1) Models exist for the purpose of eventually generating working implementations, and (2) MTL users are software engineers, developers, and architects.

At the same time, conceptual models are widely used outside of MDE. Examples include models employed in less technical subfields of Software Engineering (e.g., requirements and domain models), models used in neighboring disciplines (e.g., business process and enterprise architecture models), and models used in classical engineering disciplines. Just like the models employed in an MDE context, such models are occasionally refactored, translated, and

✉ Vlad Acreţoaie
rvac@dtu.dk

Harald Störrle
hsto@dtu.dk

Daniel Strüber
strueber@mathematik.uni-marburg.de

[1] Department of Applied Mathematics and Computer Science, Technical University of Denmark, Kgs. Lyngby, Denmark

[2] Department of Mathematics and Computer Science, Philipps-Universität Marburg, Marburg, Germany

 Springer

**Table 1** Typical skill sets of end-user modelers and MDE practitioners

|                  | Domain knowlege | Modeling | Meta-modeling | Rule languages | MT | Programming |
|------------------|-----------------|----------|---------------|----------------|-----|-------------|
| End-user modeler | ✓               | ✓        | ×             | ×              | ×   | ×           |
| MDE practitioner | ×               | ✓        | ✓             | ✓              | ✓   | ✓           |

migrated. Presently, these potential model transformation scenarios are for the most part unsupported. We argue that this lack of support is due to a mismatch between the features of existing MTLs and the requirements of end-user modelers.

### 1.1 End-user modelers

Before proposing technical solutions addressing their needs, we must first define end-user modelers and show how they differ from MDE practitioners—the target audience of most existing model transformation languages. The following definition and the ensuing discussion are based on the authors' own industrial experience and may, therefore, be questioned. It may even be argued that end-user modelers do not exist as a user category distinct from MDE practitioners, or that end-user modelers have no need for model transformation tools. These arguments can only be settled conclusively via broad empirical studies (e.g., surveys) involving practitioners. No such studies are currently available.[1]

**Definition 1** End-user modelers are non-programmer users of a modeling language familiar with its syntax and semantics, but unfamiliar with its meta-model, abstract syntax, and applicable manipulation languages.

End-user modelers are highly trained domain experts, but do not have the level of Software Engineering expertise demanded to master today's model transformation languages. Their skills typically do not include rule languages such as the Object Constraint Language (OCL [37]), meta-modeling, or computer programming using general-purpose programming languages. As illustrated in Table 1, this skill set differs significantly from that of an MDE practitioner.

End-user modelers' primary motivation is achieving meaningful domain goals with the help of models. Thus, the cost–benefit ratio of applying MTLs is a key concern for them. The learning curve imposed by a transformation language must be gentle, suggesting that MTLs should adopt modeling notations already in place in a given domain. However, end-user modelers are often knowledgeable in multiple modeling notations relevant for their domain, such as Business Process Model and Notation (BPMN [35]) and Unified Modeling Language (UML [38]) class models for business

---

[1] We are presently carrying out a large-scale practitioner survey investigating the various contexts in which models are employed. The results of this survey are as of yet unpublished.

process and information modeling, respectively. Introducing a separate MTL with custom transformation concepts for each modeling language would entail a substantial learning effort. This effort can be reduced by providing a generic MTL with consistent transformation concepts.

The second key concern for end-user modelers is their ability to understand transformations and trust the outcome of their application. Therefore, MTLs targeting this user category must be intuitively understandable. Furthermore, applying a transformation must produce predictable, reliable, and traceable results.

Finally, domain modeling places a much greater emphasis on model-to-model transformations compared to model-to-text transformations. Code generation, one of the main use cases for model transformation in MDE, loses much of its importance. Instead, use cases involving consistent global model updates, such as quality-oriented model refactoring, take center stage.

### 1.2 Transparent model transformation

To address the needs of end-user modelers, we argue that a model transformation language must be *transparent*: It must focus on what the modeler wants to achieve, rather than on technical aspects. In [2] we contend that an MTL for end-user modelers must exhibit syntax, environment, and execution transparency, concepts which we further elaborate in Sect. 3.1.

*Syntax transparency* refers to an MTL's ability to leverage the syntax of any host modeling language meeting some minimal criteria to specify transformations. A transformation specified using a syntax transparent MTL is simultaneously a valid model in its host modeling language. End-user modelers benefit from this property by not having to learn a new language for the sole purpose of specifying transformations.

*Environment transparency* indicates that an MTL does not impose restrictions on the editor used to specify transformations. It comes as a direct consequence of syntax transparency, but can also exist separately. For instance, most textual MTLs are environment transparent, as they can be used with any text editor. End-user modelers benefit from environment transparency by not having to install and learn how to use new tools.

*Execution transparency* places end-users in control of how transformations are executed by allowing them to select

the MT engine most suitable for a given task. The same specification should be executable via several engines with different capabilities, such as optimized performance or formal verification. Separating the MTL from its execution engine also helps avoid "technology lock-in" with respect to a particular modeling technology, such as the Eclipse Modeling Framework (EMF [47]). Execution transparency therefore benefits end-user modelers and MDE practitioners alike.

There are currently no MTLs targeting the needs of end-user modelers, nor are there any MTLs implementing all three aspects of transparency introduced above (see Sect. 6.2 for a review of existing MTLs implementing a subset of these aspects). We address this gap by proposing the Visual Model Transformation Language (VMTL), a usability-focused transformation language closely related to the existing, demonstrably usable Visual Model Query Language (VMQL [51]). VMTL is a model-to-model, unidirectional transformation language supporting endogenous 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 Sect. 4.3.

### 1.3 Contributions and limitations

This paper significantly extends previous work presented in [2], where VMTL and the transparent model transformation principles were first introduced. The new contributions with respect to this earlier work are:

– a definition and characterization of end-user modelers as a distinct category of model transformation language users;
– a complete description of VMTL's syntax and informal execution semantics;
– a more detailed account of tool support for VMTL;
– an empirical evaluation of VMTL's learnability, including both quantitative and qualitative methods.

VMTL is built on the groundwork laid with VMQL, a usability-oriented model query language for software models [51] and business process models [48]. Due to the fact that model transformation is in many ways a more complex operation than model querying, VMTL introduces several new language constructs. Examples include transformation rules and differentiated pattern types. VMQL's existing textual annotation language has received a general overhaul, accompanied by the introduction of transformation-specific annotations such as those supporting model element manipulation and rule execution control. In addition, the implementation of VMTL based on EMF and the Henshin model transforma-

tion engine [5] differs substantially from the Prolog-based tool support provided for VMQL [4].

Apart from defining VMTL and describing its implementation, the main contribution of this paper is the empirical validation of VMTL's learnability. Despite their importance when addressing usability-related topics, human factors evaluations such as the one presented here are relatively uncommon in model transformation research (see Sect. 6.3). That said, our evaluation suffers from some limitations which must be stated upfront. The chief threat to the validity of our user experiments is related to participant selection. Participants in both experiments were computer science students with varying degrees of modeling expertise, a background arguably different from that of end-user modelers. However, these participants also share some of the defining characteristics of end-user modelers, such as a lack of model transformation and meta-modeling experience. For a comprehensive discussion of threats to the validity of our experiments, see Sect. 5.1.4.

The think-aloud protocol analysis complementing the presented experiments benefits from a broader range of participants. However, their low number prevents us from making generalizations and drawing definitive conclusions. This and other validity concerns pertaining to this analysis are addressed in Sect. 5.2.3.

VMTL has so far not been validated on large model transformation scenarios. The sizes of the transformation specifications used in our experiments are in the same range as the examples presented in this paper. Therefore, the possibility that some of VMTL's beneficial attributes may not scale to larger transformations cannot be ruled out. Nevertheless, learnability is most critical in the early stages of adopting a new language, where small examples are more likely to be employed in the first place.

The remainder of this paper is structured as follows: Sect. 2 offers an intuitive understanding of VMTL via a running example from the banking domain, Sect. 3 presents VMTL's syntax, semantics, and limitations, Sect. 4 describes tool support, Sect. 5 presents empirical results regarding VMTL's learnability, Sect. 6 summarizes related research, and Sect. 7 presents conclusions and directions for future work.

## 2 Motivating example: model quality assurance

Quality assurance is a central concern in the life cycle of software models. The removal of *anti-patterns* (or *smells*) is therefore an important application area for model transformation tools [1]. Analysis-level models emphasize quality even further, motivated by compliance to legislation such as the Sarbanes–Oxley Act [40]. In what follows, we illustrate how quality assurance transformations on analysis-level models

can be specified using VMTL. For this purpose, consider the UML model in Fig. 1, representing a financial institution's loan operations. Models of this kind are produced by domain experts and business analysts, rather than software engineers.

The information model in Fig. 1 (top left) is expressed as a UML class diagram. It describes the two loan types offered by the financial institution: installment loans and revolving loans. Optionally, customers may purchase credit insurance. The use case model in Fig. 1 (bottom-left), expressed as a UML use case diagram, lists the interactions that a customer may have with the institution. Namely, customers can request

a loan with specified details or buy credit insurance. Finally, the process model in Fig. 1 (right), expressed as a UML activity diagram, lays out the process followed by the institution when responding to a loan request. Based on customer background and account details, a loan eligibility report and, optionally, a credit insurance offer are produced and sent to the customer.

In what follows, we introduce the core features of VMTL via three simple transformation examples, all operating on the information model in Fig. 1. A somewhat more elaborate VMTL transformation on the same model is discussed in "Appendix."
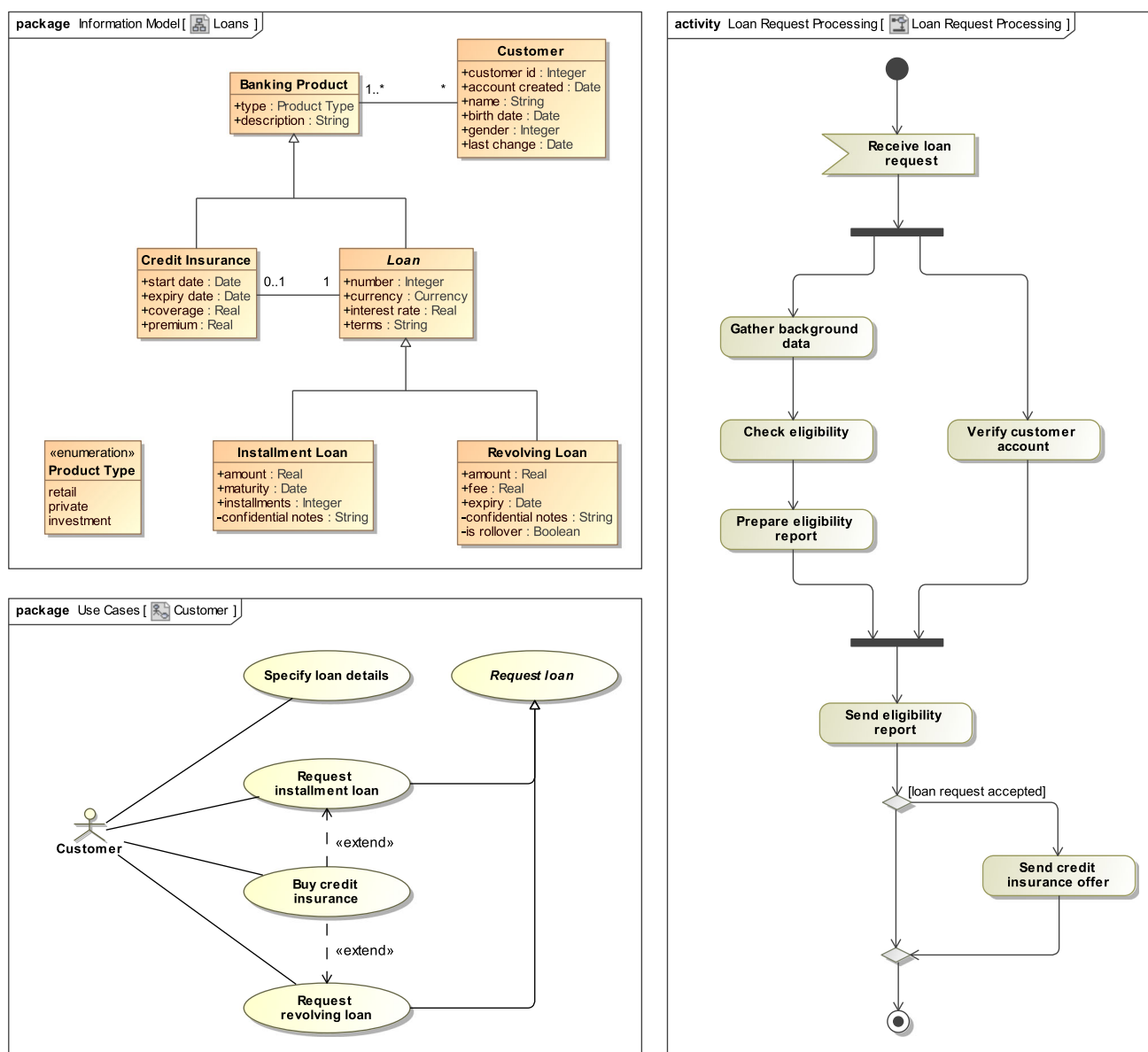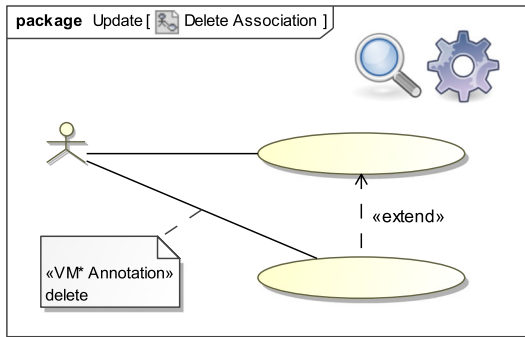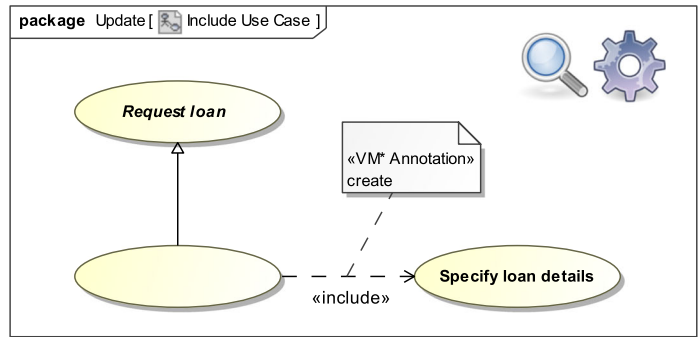


**Fig. 1** Analysis-level model representing a financial institution's loan operations. It consists of an information model expressed as a UML class diagram (*top left*), a use case model expressed as a UML use case diagram (*bottom-left*), and a process model expressed as a UML activity diagram (*right*)
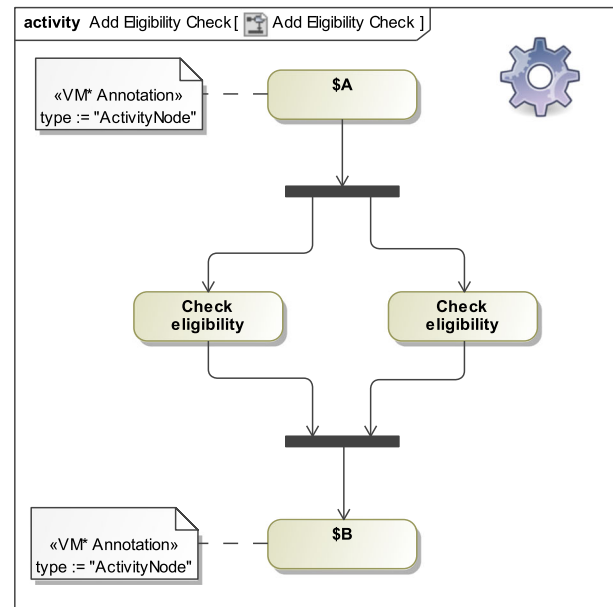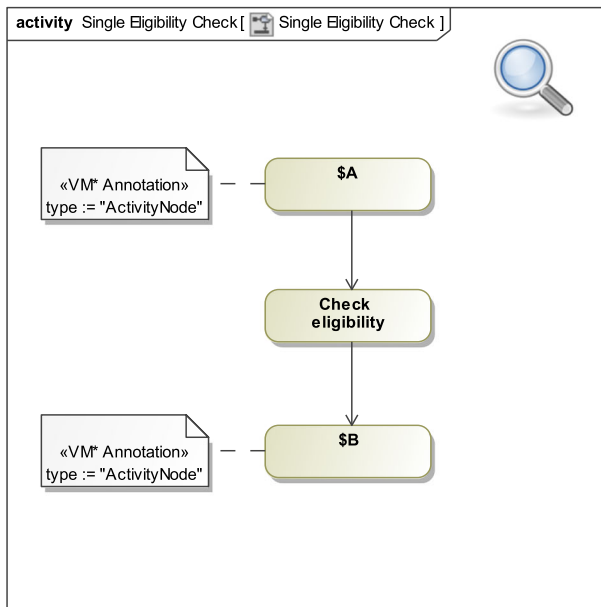
**Fig. 2** Single-rule VMTL transformations on the example source model in Fig. 1: Transformation 1 (*top left*), Transformation 2 (*top-right*), Transformation 3 (*bottom*)

## 2.1 Patterns and annotations

The "Customer" actor in Fig. 1 (bottom-left) is associated to the "Buy credit insurance", "Request installment loan", and "Request revolving loan" use cases, the last two of which extend the first. However, a UML use case extending another use case *"typically defines behavior that may not necessarily be meaningful by itself"* (see [38], p. 671). This anti-pattern can be removed by deleting the associations between the actor and the extending use cases.

Transformation 1, shown in Fig. 2 (top left), expresses this specification. It consists of a VMTL `Update Pattern` named "Delete Association." The `delete` annotation included in this pattern specifies that Associations between an actor and an extending use case must be removed from the source model. The pattern is applied twice, once for the "Request installment loan" use case and once for the

"Request revolving loan" use case. When transforming UML models, Comments are an appropriate vehicle for VMTL annotations, with the «VM* Annotation» Stereotype distinguishing these annotations from regular Comments.

VMTL patterns may optionally include visual annotations referred to as *icons*, which indicate their type. Such an annotation, expressed as a stereotyped Comment,[2] is visible in the top-right corner of the "Delete Association" pattern. While icons help visually identify pattern types, the type of a pattern is formally established by the stereotype applied to its encapsulating Package—in this case, «VM* Update».

`Update Patterns` can also create new model elements, as illustrated by Transformation 2 in Fig. 2 (top left). This transformation ensures that all loan requests require

---

[2] UML allows Stereotypes to replace model elements' default visualizations with arbitrary images.

the specification of loan details. A create annotation is employed to add an Include relationship between each use case inheriting from the "Request loan" use case and the "Specify loan details" use case.

## 2.2 Multiple-pattern rules

Financial institutions are required to ensure that critical background checks are performed several times [40]. The model in Fig. 1 violates this requirement, as the "Check eligibility" Action is performed only once. Transformation 3, shown in Fig. 2 (bottom), illustrates how VMTL can be used to duplicate this Action so that it is performed twice in parallel.

As opposed to the previous examples, this transformation is specified using two patterns: a Find Pattern and a Produce Pattern, which intuitively correspond to the "before" and "after" states of the transformation. The Find Pattern is matched in the source model, determining where the Produce Pattern is applied. In Transformation 3, the Find Pattern will match any sequence of three Actions where the middle Action is named "Check eligibility." It will do so regardless of the outer Actions' types: all instances of activity node, an abstract UML meta-class generalizing all Action types, are considered. This is accomplished via an assignment to VMTL's type *special variable*.

The $A and $B *user-defined variables* are used in the Find Pattern in place of the outer Actions' names. When the pattern is matched, these variables are instantiated with the matched Actions' names. Once instantiated, the $A and $B variables retain their values in the Produce Pattern. The differences between the two patterns determine which elements will be added to or removed from the source model. In this example, a second Action named "Check eligibility" will be created, together with a Fork Node and a Join Node. Note that this transformation could have also been specified using a single Update Pattern.

## 3 The Visual Model Transformation Language

### 3.1 A transparent approach to model transformation

Intuitively, an MTL aiming for end-user modeler accessibility should leverage languages and tools familiar to end-user modelers. This intuition forms the basis of *Transparent Model Transformation*, a collection of three general principles underlying the development of MTLs for end-user modelers: *syntax transparency*, *environment transparency*, and *execution transparency*. While a number of existing MTLs follow a subset of these principles (see Sect. 6.2), VMTL is the first to follow all three of them.

### 3.1.1 Syntax transparency

The transformation specifications in Fig. 2 do not just resemble the concrete syntax of UML diagrams. They are, in fact, valid UML models. Formally, syntax transparency is defined as follows:

**Definition 2** An MTL capable of expressing specifications for model transformations operating on source models conforming to a meta-model $\mathcal{M}$ and producing target models also conforming to meta-model $\mathcal{M}$ is said to be *syntax transparent* with respect to $\mathcal{M}$ iff all such specifications conform to $\mathcal{M}$.

Meta-model $\mathcal{M}$ is referred to as the *host meta-model* or *host language* of the transformation.

VMTL ensures syntax transparency by introducing conformance-preserving host meta-model extensions. The constructs of VMTL—rules, patterns, and annotations—are mapped to existing host meta-model elements using meta-model extension mechanisms or, if such mechanisms are not available, naming conventions. Figure 3 illustrates the realizations of VMTL constructs in UML and BPMN, respectively. The UML realizations rely on Stereotypes, such as the «VM* Update» and «VM* Annotation» Stereotypes applicable to Packages and Comments, respectively. The equivalent BPMN realizations rely on naming conventions, such as the [VM* Update] and [VM* Annotation] prefixes for Package names and Text Annotation IDs, respectively.
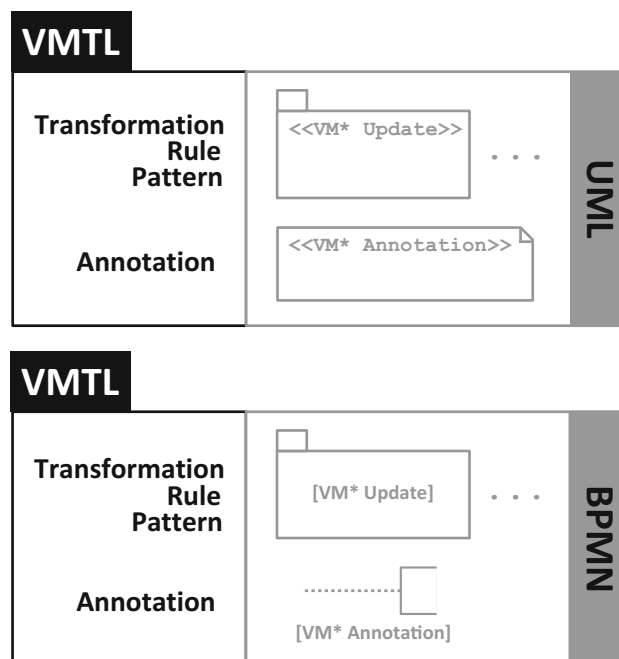


**Fig. 3** Mapping the VMTL meta-model to UML (*top*) and BPMN (*bottom*)

### 3.1.2 Environment transparency

The learning curve imposed by an MTL has two distinct contributing factors: learning the MTL itself, and learning to use the tools supporting it. The syntax transparency principle mitigates the impact of the first factor, while the second factor is addressed by the principle of environment transparency.

**Definition 3** An MTL is *environment transparent* if it allows users to adopt their preferred compatible editor for each transformation artifact: the source model(s), transformation specification, and target model(s).

Environment transparency is facilitated by syntax transparency, but can also exist independently. For instance, most textual MTLs are supported by dedicated editors, while also allowing the use of general-purpose text editors as specification tools. They therefore exhibit environment transparency. However, since specifications created using these MTLs are not valid instances of the host meta-models, textual MTLs do not exhibit 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 thus benefits end-user modelers in two respects: (1) avoiding the learning curve imposed by a new editor, and (2) leveraging a tested, mature tool. On the other hand, using the model editor to accomplish a task it was not originally designed for may bring some disadvantages, as discussed in Sect. 4.3. By promoting the loose coupling between transformation editors and execution engines, environment transparency facilitates alternative deployment avenues such as remote transformation execution, an approach likely to be beneficial in the case of large source and target models.

### 3.1.3 Execution transparency

In addition to selecting the editors of their choice, end-users should also have 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. Avoiding a tight coupling between the transformation language and its execution engine also increases the number of host modeling languages that can be supported, since different execution engines may target different modeling technologies.

**Definition 4** An MTL is *execution transparent* if specifications expressed using it can be executed by compilation to one of several *transformation engines* operating at a lower abstraction level.

A pre-condition imposed on the transformation engine is that it must support a level of expressiveness at least as high as that of the execution transparent MTL. The semantics of the MTL should not depend on the execution engine employed to implement it.

The number and complexity of language constructs included in VMTL is deliberately limited in order to facilitate its compilation to existing transformation engines. Since these constructs can be mapped to graph transformation concepts, the most intuitive compilation targets are graph transformation engines. However, implementations based on imperative engines (e.g., EOL [28]), transformation primitive libraries (e.g., T-Core [55]), or general-purpose programming languages enhanced by modeling APIs are all possible.

## 3.2 VMTL transformations: structure and execution

VMTL is a model-to-model transformation language. It supports *endogenous* transformations, that is, transformations in which the source and target models conform to the same meta-model [16,32]. VMTL transformations can be executed *in-place* to modify an existing model, as well as *out-place* to produce a new model.

VMTL specifications can be mapped to the meta-model shown in Fig. 4. According to this meta-model, a `Transformation` consists of one or more `Rules`, each having a positive integer `priority`. Rules with lower values assigned to their `priority` attribute are executed first, while rules with equal priorities are selected for execution non-deterministically. Execution terminates when no rule is applicable.

Rules consist of one or more `Patterns` expressed using the host modeling language, typically at the concrete syntax level. Patterns consist of instances of the host modeling language's elements. 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 [17]. The following pattern types are defined:

- *Find Pattern* Represents the left-hand side (LHS) of a transformation rule, specifying the source model locations at which the transformation is to be applied. A `Find Pattern` can be seen as a model query, and a rule may contain at most one such pattern. If the rule does not contain a `Find Pattern`, it must contain an `Update Pattern`.
- *Produce Pattern* Represents the right-hand side (RHS) of a transformation rule, specifying how the target model is to be obtained from the source model. A rule may contain

at most one `Produce Pattern`, and its presence is conditioned by the presence of a `Find Pattern`.

- *Update Pattern* A concise specification of both the source model locations at which a transformation is to be applied, and how the target model is to be obtained from the source model. A rule may contain at most one `Update Pattern`, under the condition that it does not contain a `Find Pattern`.
- *Require Pattern* Represents a positive application conditions (PAC) for a transformation rule. A rule can contain any number of `Require Patterns` and will be executed only if *all* of these patterns are matched in the source model.
- *Forbid Pattern* Represents a negative application conditions (NAC) for a transformation rule. A rule can contain any number of `Forbid Patterns`, and will be executed only if *none* of these patterns are matched in the source model.

Transformations, rules, and patterns can include any number of VMTL-specific textual annotations, which are also expressed as host language model elements. When anchored to a host language element included in a VMTL pattern, annotations provide additional information regarding that specific element. When anchored to a rule or to the transformation itself, annotations specify execution options, such as rule priorities or injective pattern matching. See Sect. 3.3 for descriptions of the available annotations.

Only `Find Patterns` and `Update Patterns` may *trigger* the application of a rule. The rule is triggered when one such pattern is matched in the source model, assuming that all of the rule's `Require Patterns` are also matched and none of its `Forbid Patterns` are matched. Based on

the considerations introduced so far, the full VMTL transformation execution process is illustrated as a UML activity diagram in Fig. 5. The core of this process is a rule *priority queue*.

It should be noted that this execution process does not mitigate non-terminating transformations. It is the responsibility of the end-user to ensure that VMTL transformations eventually terminate. This task is facilitated by the fact that the only execution control mechanisms supported by VMTL are rule priorities and application conditions. Some basic heuristics can be applied to determine if a risk of non-termination exists. The most important such heuristic is the presence of a `create` clause within a given rule. In the absence of a `Forbid Pattern`, a `Require Pattern`, or an `omit` clause limiting its applicability, the creation of new model elements can continue without termination. On the other hand, VMTL rules including the `delete` clause in the absence of any `create` clauses are guaranteed to terminate. As a good practice, we encourage the use of `Forbid Patterns` and `Require Patterns` to make rule application conditions explicit, even in cases where they could be embedded as clauses in a `Find Pattern` or an `Update Pattern`.

Based on the presented meta-model and execution process, we argue that VMTL is theoretically and practically expressive enough for most endogenous transformation scenarios faced by end-user modelers. First, the constructs of VMTL (i.e., the meta-classes in Fig. 4) are a common subset of the constructs found in current mainstream transformation languages. This is a deliberate choice that promotes execution transparency. The subset is minimal in the sense that removing any of the constructs would bring obvious practical expressiveness limitations. In terms of theoretical
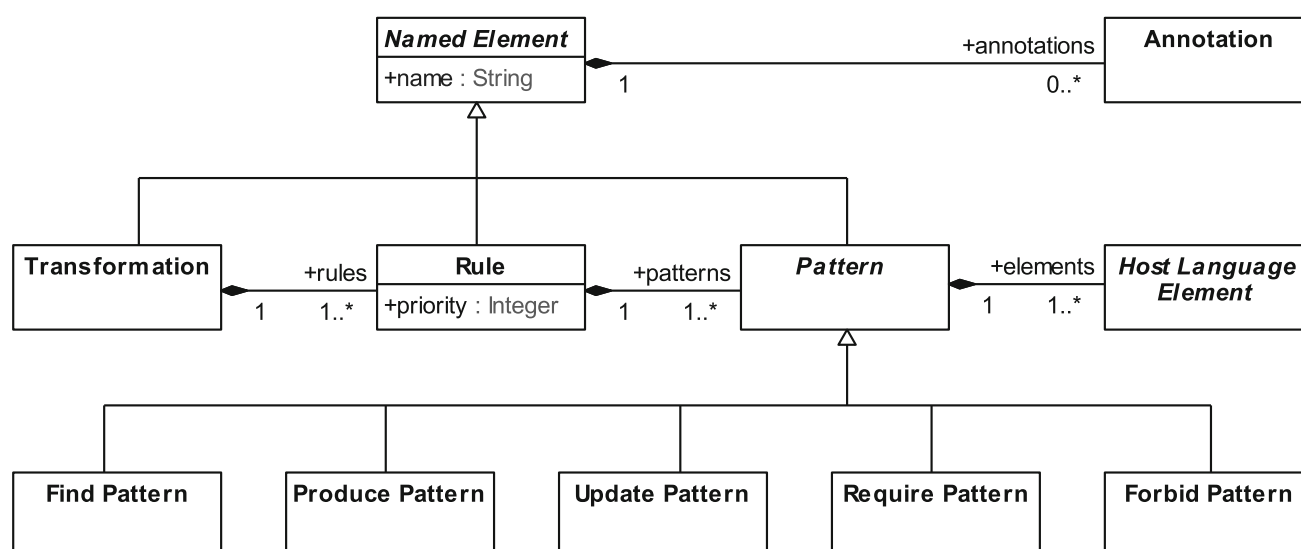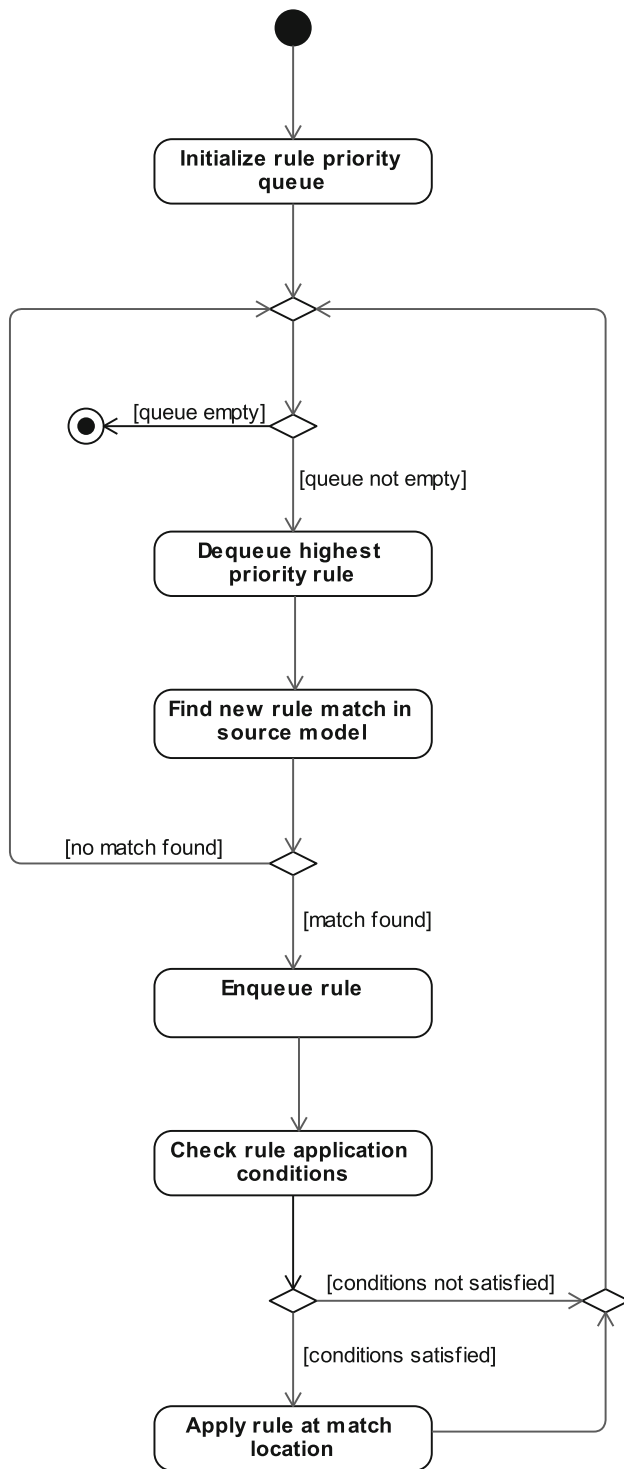


**Fig. 4** The VMTL meta-model

**Fig. 5** The VMTL transformation execution process

expressiveness, we point out that the execution semantics of VMTL shown in Fig. 5 is implicitly recursive, while `Forbid Patterns` and `Require Patterns` allow the expression of branching constructs. Together with support for user-defined variables, these features likely make VMTL

Turing complete, although a formal proof of this property remains as future work.

Following the principle of syntax transparency, the VMTL meta-model elements in Fig. 4 are mapped to host meta-model elements. The mapping can be implemented using the host metamodel's extension mechanism, or, if such a mechanism is not available, using model element naming conventions. Meta-model extension mechanisms are the more systematic, and therefore preferred solution. As an example of such a mapping, the VM* Profile for UML, defined in Table 2, includes a Stereotype for each VMTL meta-model element.

Among the stereotypes listed in Table 2, those identifying VMTL patterns can be applied to both Packages and Comments. However, package stereotypes are sufficient for defining a transformation's structure. These stereotypes may optionally be applied to comments displayed in diagrams in order to visually indicate the type of VMTL pattern contained in a Package. When such a Stereotype is applied to a comment, the standard UML comment notation is replaced by the corresponding icon shown in Table 2.

The optional pattern icons provide an intuitive way of identifying the type of a pattern. A magnifying glass indicates the "search" functionality of a `Find Pattern`, while a cogwheel hints at the model modifications performed by a `Produce Pattern`. Since `Update Patterns` merge the functionality of `Find/Produce` pattern pairs, the icon for `Update Patterns` is also a merger of these patterns' icons. The icons for `Forbid Patterns` and `Require Patterns` employ common symbols (an "access forbidden" sign and a checkmark) and color coding.

In general, VMTL transformations can only be specified if the host meta-model meets the following prerequisites, the first two of which are mandatory:

1. The host meta-model must include a *container element*, such as Packages in UML and BPMN. Container elements are used to structure VMTL transformations into rules and patterns.
2. The host meta-model must include an *annotation element*, such as Comments in UML and Text Annotations in BPMN. All host meta-model elements must support annotations, which act as vehicles for VMTL clauses.
3. The host meta-model should support a lightweight extension mechanism, such as UML Stereotypes, allowing the identification of model elements as VMTL constructs. Such a mechanism is optional, and can be substituted by element naming conventions.

### 3.3 Annotation syntax

VMTL patterns are valid models in the host modeling language. Although it supports syntax and environment trans-

**Table 2** The VMTL profile for UML. The VMTL profile for UML. This profile is applied to UML packaged containing VMTL transformation specifications

| Stereotype | Applies to | Description | Icon |
|---|---|---|---|
| «VM* Annotation» | Comment | Stereotype applicable to Comments containing VMTL annotations | – |
| «VM* Transformation» | Package | Stereotype applicable to Packages containing a VMTL transformation | – |
| «VM* Rule» | Package | Stereotype applicable to Packages containing a VMTL rule | – |
| «VM* Find» | Package, Comment | Stereotype applicable to Package containing a Find Pattern or to Comments included in such Packages, in which case the Find icon replaces the UML Comment notation | |
| «VM* Produce» | Package, Comment | Stereotype applicable to Packages containing a Produce Pattern or to Comments included in such Packages, in which case the Produce icon replaces the UML Comment notation | |
| «VM* Update» | Package, Comment | Stereotype applicable to Packages containing an Update Pattern or to Comments included in such Packages, in which case the Update icon replaces the UML Comment notation | |
| «VM* Forbid» | Package, Comment | Stereotype applicable to Packages containing a Forbid Pattern or to Comments included in such Packages, in which case the Forbid icon replaces the UML Comment notation | |
| «VM* Require» | Package, Comment | Stereotype applicable to Packages containing a Require Pattern or to Comments included in such Packages, in which case the Require icon replaces the UML Comment notation | |

parency, this design decision has the potential of severely limiting the expressiveness and usefulness of VMTL. The root cause of this is the fact that modeling languages are not designed to support pattern specifications—and they have no reason to do so. Therefore, specifying VMTL patterns sometimes requires sidestepping well-formedness constraints included in the host meta-model (e.g. element multiplicity limits), as well as referring to elements of the VMTL meta-model (e.g., for specifying execution options and transformation rule priorities). Kühne et al. [29] propose achieving this by explicitly modifying, or *relaxing*, the host meta-model. VMTL does not adopt this approach, as it would violate syntax and environment transparency.

Instead, under the assumption that existing elements of the host meta-model must not be modified, VMTL defines a simple *textual annotation language* used to "lift" models to the status of model patterns. From an end-user modeler perspective, learning this annotation language is the only significant prerequisite of using VMTL. The defined annotations support pattern definition, model manipulation, and transformation execution control. As examples, several VMTL annotations are included in Figs. 2 and 14 as UML Comments carrying the «VM* Annotation» Stereotype.

The specification of this annotation language is one of the most sensitive aspects of VMTL's overall design. To begin with, adopting an annotation syntax resembling that of a widely used general-purpose programming language such as Java or JavaScript brings no benefits in an end-user modeler context. These languages are unfamiliar and too complex for end-user modelers. A second option would be to use

an "off-the-shelf" annotation language, such as OCL. However, OCL has been shown to be severely lacking in terms of usability, especially when compared to natural language-like alternatives [49]. We have therefore decided to equip VMTL with its own textual annotation language. This language is loosely based on logic programming principles: an annotation consists of a set of *clauses*, each expressing a constraint on the annotated pattern elements or the VMTL specification as a whole. Clauses are tied together by logic operators. Our hypothesis, based on previous findings regarding VMQL [48,51], is that such a mechanism is both sufficiently expressive and easy to learn by end-user modelers. Its features are detailed in the following paragraphs.

Dynamically typed *user-defined variables* may be declared and manipulated in VMTL annotations, and also used as meta-attribute values. The *scope* of these variables is limited to a single rule application: Once declared, their value can be accessed in all patterns belonging to the applied rule, but not in patterns belonging to other rules. Due to their rule-wide scope, user-defined variables are employed for identifying corresponding model elements across a rule's patterns. User-defined variable's names are prefixed by the $ character.

The type of a user-defined variable is inferred at rule execution time. VMTL supports the Boolean, Integer, Real, and String data types, in addition to the Element data type used for storing instances of host language meta-classes. Regardless of their type, user-defined variables also accept the *undefined value* ("*"). A variable with this value is interpreted as representing any valid value of its respective data type.

**Table 3** Special variables supported in VMTL annotations. The type, scope and description of each special variable is provided, accompanied by usage examples

| Variable | Type | Scope | Description | Examples |
|---|---|---|---|---|
| `id` | String | Element | Stores an optional user-defined pattern element identifier in order to facilitate the identification of corresponding elements across patterns | `id := "1"` |
| `injective` | Boolean | Rule | If set to `true` (the default value), each pattern element can be matched to at most one source model element. Otherwise, each pattern element can be matched to several source model elements | `injective := true` |
| `priority` | Integer | Rule | Determines the application priority of a rule. Only positive values are allowed, with lower values implying a higher execution priority | `priority := 1` |
| `self` | Element | Element | Allows access to the annotated model element. | `self.visibility := "public"` |
| `steps` | Integer | Element | States that the annotated model element, which must represent a relation, can be matched to a chain of relations of the same type in the source model. The length of the chain is determined by the value of this special variable | `steps := 3, steps > 3, steps := *` |
| `type` | String | Element | Provides access to the name of the annotated model element's meta-class. Assigning a new value to this special variable modifies the annotated model element's meta-class | `type := "Actor"` |

For variable manipulation, VMTL supports arithmetic, comparison, and logic operators. Logic operators can be invoked through textual notations ("and", "or", "not", "if/then") or logic programming-style notations (",", ";", "!", "->"). The implication and disjunction operators can be combined to form a conditional "if/then/else" construct (see Rule 1 of the refactoring in "Appendix" for example). The navigation operator (".") accesses model elements' meta-attributes, operations, and association ends.

Apart from user-defined variables, VMTL relies on *special variables* as a means of controlling transformation execution (the `injective`, `priority`, and `steps` variables) and accessing the contents of the source model (the `id`, `self`, and `type` variables). The special variables defined by VMTL are listed in Table 3. All operators applicable to user-defined variables are also applicable to special variables. As an example, the `type` special variable is used to specify the meta-type of several model elements in Transformation 3 (Fig. 2). Because `ActivityNode` is an abstract meta-type, it cannot be assigned to UML model elements. However, as this example shows, assigning this meta-type to pattern elements may be required and can be achieved via VMTL's `type` annotation.

Special variables have a pre-defined `scope`, identifying the specification fragment to which they are applicable. For instance, the scope of the `priority` special variable, which represents the mechanism used to specify rule priority in VMTL, is limited to one rule, while the scope of the `id`, `self`, and `type` variables is limited to the annotated model element.

*Clauses* are the main building blocks of VMTL annotations: each annotation consists of one or more such clauses connected by logic operators. The use of clauses is inspired by logic programming languages and adopted as a means to achieve annotation conciseness. A VMTL clause is an assertion made about the pattern model elements to which its containing annotation is anchored, about its containing pattern or rule as a whole, or about user-defined or special variables. Some clauses specify modifications to the source model (the `create` and `delete` clauses), while others act as constraints on pattern matching (the `either`, `indirect`, `omit`, `optional`, and `unique` clauses). Variable assignment (":=") is also treated as a clause. The clauses included in VMTL's annotation language are listed in Table 4.

Notably, the `either` clause can only be included in annotations anchored to several pattern model elements. All other clauses listed in Table 4 can be included in annotations anchored to one or more pattern elements. In general, anchoring a clause to several pattern elements instead of creating several annotations containing the same clause allows more compact specifications. The variable assignment clause (":=") can also appear un-anchored to any pattern elements, as variables always have a rule-wide scope.

## 4 Tool support

While VMTL as a language is syntax transparent, its environment and execution transparency depend on its implementa-

**Table 4** Clauses supported in VMTL annotations. A description and a list of pattern types in which it can be applied is provided for each clause

| Clause | Description | Patterns types |
|---|---|---|
| := | Assigns a value to a user-defined variable, special variable, or model element meta-attribute | Find, Produce, Update, Forbid, Require |
| create | Creates the annotated pattern model element in the target model. If a model element not included in the Find Pattern of a rule is included in the rule's Produce Pattern, the element is implicitly created in the target model. In such cases, the create clause is optional | Produce, Update |
| create if not exists | Creates the annotated pattern model element in the target model only if it does not exist in the source model | Produce, Update |
| delete | Deletes the annotated pattern model element from the source model. If a model element included in the Find Pattern of a rule is not included in the rules's Produce Pattern, the element is implicitly deleted in the target model. In such cases, the delete clause is optional | Produce, Update |
| either | Specifies that exactly one of the annotated pattern model elements must be matched in the source model | Find, Forbid, Require |
| indirect | Specifies that the annotated pattern model element, which must represent a relation, can be matched to a chain of relations of the same type (i.e., the relation's transitive closure) in the source model | Find, Forbid, Require |
| omit | Specifies that the annotated pattern model element must not be matched in the source model | Find, Forbid, Require |
| optional | Specifies that the annotated pattern model element may or may not be matched in the source model | Find, Forbid, Require |
| unique | Specifies that the annotated pattern model element must be unique within its scope (e.g., its containing Package) in the source model. When this annotation is included in an Update Pattern, the uniqueness condition is applied to both the source and the target model | Find, Produce, Update, Forbid, Require |

tion. As detailed in this section, our implementation follows all three Transparent Model Transformation principles.

### 4.1 Executing VMTL specifications

The implementation of VMTL is based on the Eclipse Modeling Framework (EMF [47]) and the Henshin [5] transformation engine. The core of this implementation[3] was developed over a period of three months by the first author, with no previous experience using the Henshin engine. Henshin was adopted due to the fact that its graph transformation-based operational semantics aligns well with the semantics of VMTL. As a stand-alone API, it also supports VMTL's environment transparency. However, following the principle of execution transparency, any sufficiently expressive transformation engine could be used instead. Such an engine must meet the following minimal requirements:

- *Rules:* Self-contained execution units that can be mapped to VMTL rules must be supported.
- *Pattern matching:* The engine must support a mechanism for expressing model patterns and matching them on models.

- *Variables:* Either statically or dynamically typed variables with user-defined names and values must be supported.

Additional features such as a rule scheduling mechanism and rule application conditions can be implemented on top of the execution engine. In our case, Henshin already supports these features. Ultimately, VMTL could be implemented using a general-purpose programming language, much like its predecessor VMQL was implemented using standard Prolog. However, execution transparency encourages the reuse of existing transformation engines.

Translating VMTL specifications into a format compatible with the underlying execution engine requires a one-time software development effort. This is obviously not a task for the end-user modeler, but one for a tool provider. The development effort and skills required from this provider are a function of the selected execution engine: the closer its constructs and semantics to those of VMTL, the less effort is required.

The high-level architecture of our implementation, shown in Fig. 6, consists of three components. A general-purpose model editor is used to create the source model and transformation specification, as well as view the target model. The Henshin engine applies the transformation to the source
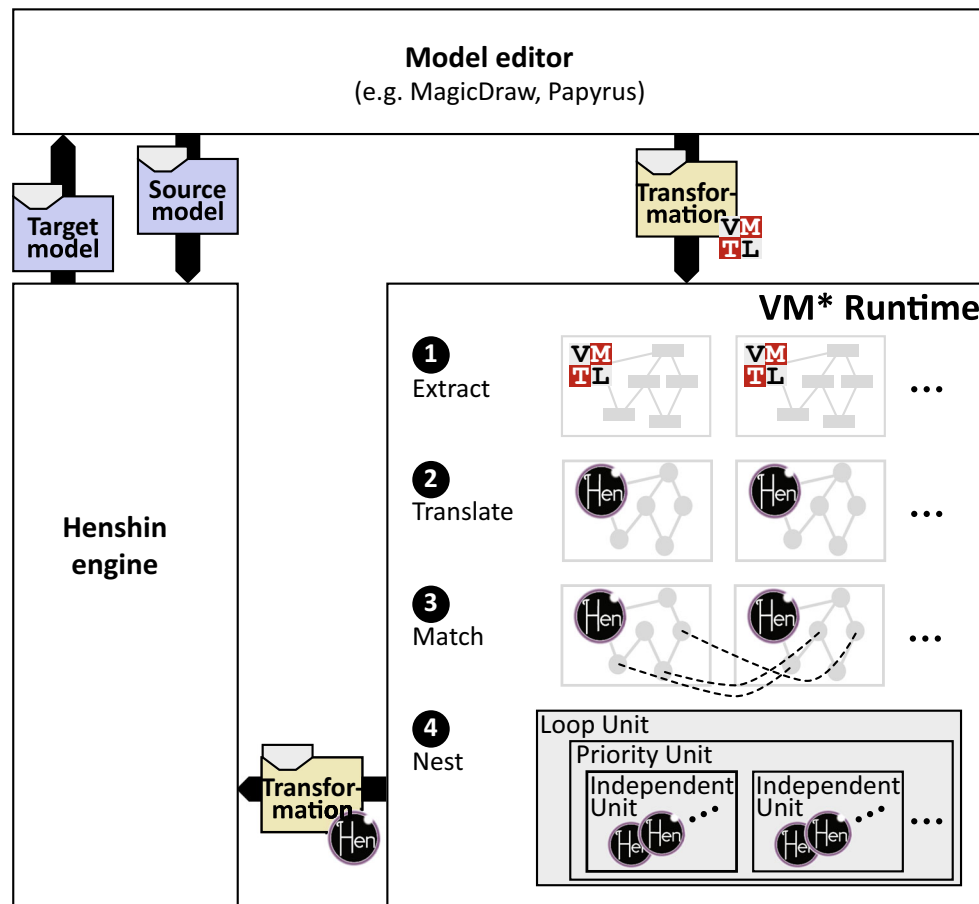
---

[3] Only a subset of the textual annotations described in Sect. 3.3 are currently supported.

**Fig. 6** The architecture of VMTL's implementation. *Numbers* encircled in *black* indicate the sequence of steps in the VMTL to Henshin compilation process

model, producing the target model. The VM* Runtime—the only component of this architecture created specifically for the purpose of supporting VMTL—compiles VMTL specifications into equivalent Henshin specifications. The compilation performed by the VM* Runtime can be seen as a higher-order transformation (HOT), the four steps of which are shown in Fig. 6 and described in what follows. The first step is transformation engine-independent, while the last three steps are specific to the Henshin engine and would need to be re-implemented to accommodate a different transformation engine.

In step ① model fragments representing transformation components are identified in the VMTL specification. These are the transformation's left-hand side (LHS), right-hand side (RHS), negative application conditions (NAC), and positive application conditions (PAC). As the components correspond to VMTL rules and patterns, their identification is informed by VMTL stereotypes or naming conventions.

In step ② 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. To avoid binding the implementation to

a particular modeling language, the fragments are processed in terms of the Ecore meta–meta-model. Thus, the task at hand is to perform an exogenous transformation between an Ecore model instance and a Henshin graph instance. This transformation is facilitated by the fact that Henshin model elements (e.g., `Node`, `Edge`, and `Attribute`) maintain explicit references to corresponding Ecore model elements (e.g., `EClass`, `EReference`, and `EAttribute`).

In step ③ 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 EMF Compare model comparison framework [34] is used for match computation. In order for the generated Henshin rules to have the expected behavior, the computed match must be exact. The use of VMTL's `id` special variable to uniquely identify unnamed pattern elements across patterns guarantees an exact match.

In step ④ the generated rules are nested in `Units`, Henshin's control flow mechanism. Each Henshin rule is assigned
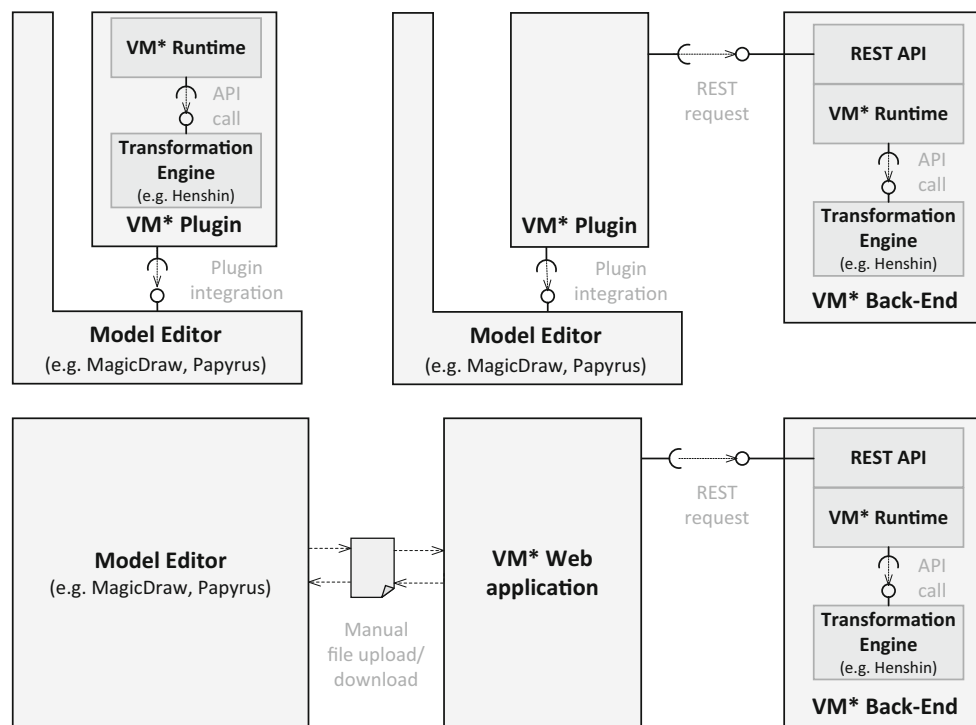
**Fig. 7** Deployment options available for VMTL: a self-contained model editor plug-in (*top left*), a thin client model editor plug-in (*top-right*), and a web application (*bottom*)

the priority of the VMTL rule from which it was derived. First, all rules with the same priority are nested inside a single `Independent Unit`, allowing non-deterministic rule selection. Next, all `Independent Units` are assigned as subunits to a `Priority Unit`, ensuring that the highest-priority `Independent Unit` is executed. Finally, the `Priority Unit` is encapsulated in a `Loop Unit`, so that it is executed as often as it is applicable. The resulting control structure implements the operational semantics of VMTL: the highest-priority applicable rule is executed until no applicable rules exist, at which point the transformation terminates.

### 4.2 Deployment options

The architecture presented in Fig. 6 is amenable to several deployment options. In a monolithic plugin-based deployment, illustrated in Fig. 7 (top left), a VMTL plugin for a conventional model editor encapsulates both the VM* Runtime and the MT engine. This is arguably the most widespread deployment approach adopted by MT tools today, many of which, including Henshin, are developed as full-featured plugins for the Eclipse IDE. However, this approach offers limited portability, since a separate plugin must be developed for every model editor that is not based on the Eclipse platform.

To improve portability without sacrificing editor integration, the VM* Runtime and the MT engine can be deployed remotely and accessed via a REST API,[4] as shown in Fig. 7 (top-right). This way, business logic is removed from the editor plugin, facilitating its re-implementation. On the other hand, this type of distributed deployment brings a number of inherent drawbacks. Transferring large models over a network may become a performance bottleneck, while remote model processing requires sound access control provisions.

A third option, illustrated in Fig. 7 (bottom), is to forego editor integration entirely and develop a separate web application as a user interface for VMTL. This solution allows specifying VMTL transformations using any editor supporting the host modeling language, without requiring a custom plug-in. The cost is that users must leave the model editor in order to apply the transformations, making interactive transformation execution infeasible. The already mentioned issues related to remote model processing also apply. We have previously demonstrated that such an approach is viable by adopting a service-oriented deployment for the Hypersonic model analysis API. A more in-depth analysis of its advantages and drawbacks is available in [3].

The deployment option in Fig. 7 (bottom) is the only one to strictly follow the environment transparency principle,

---

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

as it does not require any customization of the host model editor. Our ongoing implementation thus targets this deployment strategy, with the mention that once the VM* Back-End is operational, lightweight editor plugins implementing the architecture in Fig. 7 (top-right) can be created with relatively little effort. We currently target the MagicDraw[5] modeling environment with such a plug-in.

Arguably the most important concern that requires mitigation in our distributed deployment is unauthorized model access. Namely, it must be guaranteed that users cannot gain unauthorized access to models uploaded by other users. When combined with a role-based authentication policy, a sound authentication mechanism such as OAuth [22] is an effective method of providing this guarantee. At a technical level, implementing OAuth requires VM* API clients to obtain an access token prior to using the API. This process can be carried out through a separate channel, such as a dedicated API management web application.

### 4.3 Tool support limitations

The implementation of VMTL follows the principles of environment and execution transparency, thus facilitating its adoption by end-user modelers. However, following these principles also brings some limitations.

As a consequence of execution transparency, possible incompatibilities between VMTL's operational semantics and the capabilities of its underlying MT engine should be considered. 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 engines, but trivial for, say, a logic programming-based engine.

In the context of environment transparency, model editors are employed for a task they were not designed for—specifying transformations. In the case of VMTL, the well-formedness and syntactical correctness of transformation rules cannot be verified inside the editor in the absence of a dedicated plugin. Most model editors will, however, enforce the conformance of VMTL patterns to the host meta-model. The resulting expressiveness limitation is mitigated by VMTL's textual annotations. At execution time, transformation tracing and debugging must be performed through an editor extension or outside the model editor, such as through the web application described as a deployment option in Sect. 4.2. Finally, displaying target models in the host editor is complicated by the fact that diagram layout information is typically not part of the host meta-model. Maintaining a layout similar to that of the source model is therefore only possible for in-place transformations.

To preserve environment transparency, VMTL does not support explicit mappings between the elements of different patterns included in a transformation rule. Instead, the VM* Runtime infers the mappings as described in Sect. 4.1. 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. This may lead to ambiguities when a transformation is executed. VMTL addresses this issue at the language level, by providing the `id` special variable to attach an optional identifier to each pattern element. It is the transformation developer's responsibility to ensure that corresponding elements have the same identifier in all patterns.

Finally, VMTL's declarative nature may cause problems regarding rule application and transformation termination. Two transformation rules are in conflict if one of them modifies the source model in a manner that affects the applicability of the other. Furthermore, some VMTL transformations might fail to terminate. For example, any transformation adding elements to a model without imposing application conditions falls in this category. In such cases, the underlying MT engine can support the VMTL transformation developer by formally analyzing specifications. The Henshin engine supports *critical pair analysis*, a technique originating in graph transformation theory [13]. However, this technique has limitations: The termination of a graph transformation system is undecidable in the general case.

## 5 Evaluation

We have experimentally evaluated the learnability of VMTL. The methodology and outcomes of this evaluation are presented in Sect. 5.1. As a follow-up, we have conducted a think-aloud protocol analysis aiming to detect shortcomings in the language and discover how users comprehend VMTL specifications. The results of this investigation are reported in Sect. 5.2.

### 5.1 Learnability experiments

#### 5.1.1 Methods and materials

The purpose of our experiments was to evaluate the *initial learnability* [20] of VMTL, i.e., user's initial performance when first faced with the language. This property is arguably important in the context of end-user model transformation, as most end-user modelers have no prior experience with MT technology.

---

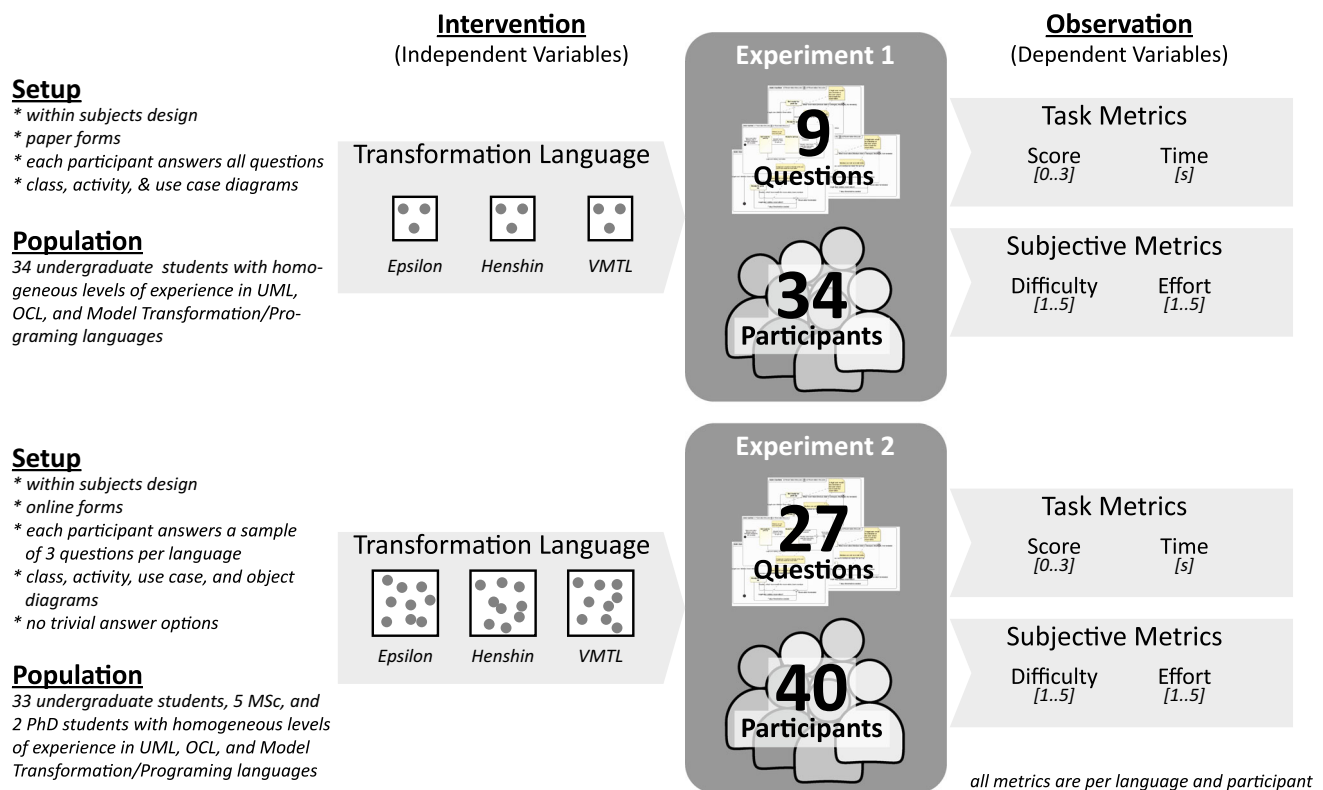5 http://www.nomagic.com/products/magicdraw.

**Fig. 8** Overview of the study designs adopted for the learnability experiments

To this end, we have carried out two questionnaire-based experiments, both of which compare the initial learnability of VMTL with that of a textual MTL (Epsilon) and a visual abstract syntax MTL (Henshin). Epsilon and Henshin were selected as representing the main transformation paradigms in the MDE landscape: textual imperative/declarative hybrid languages, and visual graph transformation-based languages [16]. In addition, Epsilon and Henshin are in widespread use.[6]

The two experiments, referred to in what follows as *Experiment 1* and *Experiment 2*, differ in terms of design and the difficulty of included tasks. These differences are highlighted in the following paragraphs. The two experimental setups are summarized in Fig. 8. A replication package containing the questionnaires, statistical analysis scripts, and the collected data is available online [63].

Both experiments are *crossover studies*, a variant of *within-subject design* [23]: all participants were sequentially exposed to each MTL. The crossover design was selected due to its statistical efficiency, as it minimizes the number of participants required to correctly identify statistically significant differences between the MTLs. The main threats to the validity of our experiments are related to participant selection, learning effects, and possible participant bias in favor of VMTL as a language developed by their teachers. The mitigation measures adopted against these threats are described in Sect. 5.1.4.

Experiment 1 took place in Spring 2014 and included 34 bachelor level CS students as participants. Experiment 2 took place in Spring 2015 and included 40 bachelor, master, and doctoral level CS students. In a subjective self-evaluation, over 80 % of participants rated their own knowledge of UML and programming as good or very good, and their knowledge of OCL and MT as poor or very poor. With the exception of participants' programming skills, these ratings are consistent with the skills of an end-user modeler (see Table 1).

Immediately before the experiments, participants were offered a written handout containing a brief introduction to MT and descriptions of the three MTLs. They were asked to read the handout and allowed to consult it at any time during the experiment. Participants were then presented with a questionnaire consisting of two sections: a *comprehension* section and an *assessment* section. Different questionnaires were used in Experiment 1 and Experiment 2.

The comprehension sections of both experiments' questionnaires contain nine multiple-choice questions, three for every MTL. To answer a question, participants were required to select the correct natural language description of a given

---

[6] A list of Epsilon's industrial users is available at https://www.eclipse.org/epsilon/users/. A collection of publications describing Henshin's use in various projects is available at https://www.eclipse.org/henshin/publications.php.

**Table 5** Mean ($\mu$), median ($M$), and standard deviation ($\sigma$) of the sizes of transformation specifications used in comprehension questions. For Epsilon, size is measured by counting lines of code. For Henshin and VMTL, size is measured by counting diagram elements, as described in [50]

|  | Specification size | | |
|---|---|---|---|
|  | $\mu$ | $M$ | $\sigma$ |
| *Experiment 1* | | | |
| Epsilon | 18.00 | 15 | 4.24 |
| Henshin | 22.67 | 27 | 8.34 |
| VMTL | 26.00 | 28 | 4.32 |
| *Experiment 2* | | | |
| Epsilon | 18.44 | 16 | 8.37 |
| Henshin | 78.00 | 68 | 33.40 |
| VMTL | 21.22 | 21 | 5.49 |

**Table 6** Natural language description options provided in Experiment 2 for Transformation 1 in Fig. 2. The correct answer option is (a)

a. If two use cases are associated with the same actor, and one of the use cases extends the other, delete the association between the actor and the extending use case

b. If a use case extends another use case, delete all actors associated with the extended use case

c. If two use cases are associated to the same actor, and one of the use cases extends the other, delete the association between the actor and the extended use case

d. I don't know

MT specification from a set of three answer options. In Experiment 1, each participant was presented with the same transformation three times, once for every MTL. In Experiment 2, participants were presented with each transformation only once, with a balanced number of participants receiving each transformation specified in each MTL. The MT specifications included in Experiment 1 operate on UML class, use case, and activity diagrams. In addition to these diagram types, Experiment 2 also includes specifications operating on UML Object diagrams. The comprehension section produces the experiments' task metrics: the *comprehension score*, i.e., the number of correct answers provided by a participant for each MTL (ranges between 0 and 3), and the *time* required by a participant to answer the questions for each MTL.

The sizes of the transformation specifications included in our experiments are summarized in Table 5. The size of Epsilon specifications is measured by counting lines of code, while the size of Henshin and VMTL specifications is measured by counting individual shapes, line segments, and textual labels (see [50] for a discussion of diagram size metrics). Questions included in Experiment 2 address slightly more complex transformations and offer at least two plausible answer options per question, leading us to replace the Epsilon Transformation Language (ETL) with the closely related but less constraining Epsilon Object Language (EOL).

The example transformations in Figs. 2 and 14 ("Appendix") are representative for the type of transformations included in our experiments, namely in-place model updates as commonly used for software model quality assurance [1]. Transformation 1 in Fig. 2 is in fact used as a question in Experiment 2, which also includes equivalent questions formulated for Epsilon and Henshin. The answer options presented to participants for this question are listed in Table 6.

The assessment section of the questionnaires addresses participants' subjective evaluation of the *cognitive load* imposed by each MTL. Two metrics were collected using Likert scales: *difficulty* and *effort* ratings. Complementary qualitative information regarding cognitive load was collected via follow-up interviews with some of the participants.

To facilitate evaluating the effect of the MTLs on participants with different skill and capability levels, we analyze data originating from high-performing and low-performing participants separately. The average comprehension score is used as a threshold value for identifying high performers and low performers.

We rely on the analysis of variance (ANOVA [33]) as a statistical hypothesis testing approach. The variance homogeneity and normal distribution of observations required as prerequisites for applying ANOVA were verified, as recommended in the literature [33], using residual plots and Q-Q plots. We also employ the Wilcoxon signed-rank test as a non-parametric alternative, thus strengthening our confidence in the observed ANOVA results. Effect sizes are evaluated using the $\eta^2$ statistic in the case of ANOVA,[7] and Spearman's $\rho$ in the case of the Wilcoxon signed-rank test.[8]

### 5.1.2 Observations

The means and standard deviations of the comprehension scores resulting from Experiment 1 and Experiment 2 are presented in the leftmost data columns of Table 7 and Table 8, respectively. The scores are also visualized as stacked bar graphs in Fig. 9. Each horizontal bar in the figure is split into sections corresponding to possible comprehension scores. The size of each section is proportional to the number of participants which have obtained that particular score. Since in both experiments the comprehension section consists of three questions for each transformation language, possible comprehension scores range between 0 and 3. In Fig. 9,

---

[7] Guidelines suggest that $\eta^2$ values greater than 0.06 indicate a moderate effect size, and values greater than 0.14 indicate a large effect size [15].
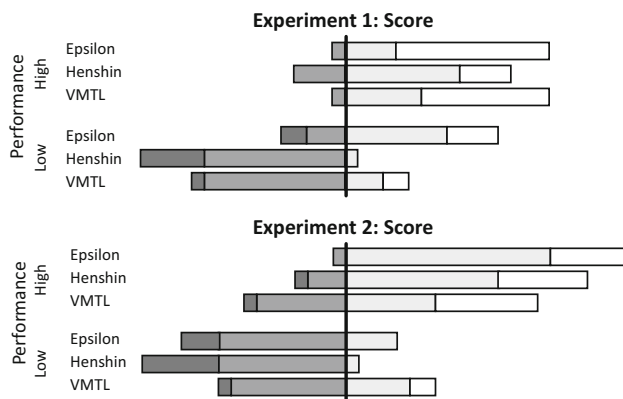
[8] Values for Spearman's $\rho$ range in the interval $[-1, 1]$. Values closer to 0 indicate a lower correlation, and therefore a larger effect size.

**Table 7** Mean ($\mu$) and standard deviation ($\sigma$) of comprehension scores, comprehension times, and cognitive load ratings for Experiment 1

| | Score [0..3] | | Time (s) | | Difficulty [1..5] | | Effort [1..5] | |
|---|---|---|---|---|---|---|---|---|
| | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| *High performers* | | | | | | | | |
| Epsilon | 2.65 | 0.60 | 325.13 | 110.74 | 3.93 | 0.79 | 3.61 | 0.86 |
| Henshin | 2.00 | 0.71 | 240.35 | 105.72 | 3.95 | 0.83 | 3.54 | 0.78 |
| VMTL | 2.53 | 0.63 | 275.41 | 123.93 | 4.22 | 0.75 | 3.65 | 0.88 |
| *Low performers* | | | | | | | | |
| Epsilon | 1.82 | 0.95 | 268.41 | 137.72 | 3.11 | 0.68 | 2.97 | 0.34 |
| Henshin | 0.76 | 0.56 | 245.00 | 106.84 | 3.28 | 0.73 | 3.11 | 0.68 |
| VMTL | 1.35 | 0.79 | 245.47 | 135.86 | 3.05 | 0.77 | 3.05 | 0.82 |

**Table 8** Mean ($\mu$) and standard deviation ($\sigma$) of comprehension scores, comprehension times, and cognitive load ratings for Experiment 2

| | Score [0..3] | | Time (s) | | Difficulty [1..5] | | Effort [1..5] | |
|---|---|---|---|---|---|---|---|---|
| | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| *High performers* | | | | | | | | |
| Epsilon | 2.22 | 0.52 | 441.00 | 180.81 | 4.06 | 0.87 | 4.06 | 0.94 |
| Henshin | 2.09 | 0.79 | 495.65 | 295.17 | 4.39 | 0.70 | 4.28 | 0.83 |
| VMTL | 1.96 | 0.93 | 326.09 | 115.59 | 3.78 | 0.81 | 3.56 | 0.86 |
| *Low performers* | | | | | | | | |
| Epsilon | 1.06 | 0.66 | 420.00 | 136.38 | 2.95 | 1.00 | 3.41 | 1.14 |
| Henshin | 0.71 | 0.59 | 351.43 | 107.48 | 2.86 | 1.08 | 3.00 | 1.07 |
| VMTL | 1.47 | 0.80 | 402.86 | 235.84 | 2.73 | 1.03 | 2.55 | 1.14 |



**Fig. 9** *Stacked bar graphs* illustrating comprehension scores for each MTL. *Lighter colors* correspond to higher scores: *white* sections show the proportion of participants obtaining the maximum score (3), *dark grey* sections show the proportion of participants obtaining the minimum score (0) (color figure online)

lighter colors correspond to higher scores: white sections represent the proportion of participants that have obtained the maximum score (3), while dark grey sections represent the proportion of participants that have obtained the minimum score (0). All plots in the figure are centered by a vertical line drawn between the sections corresponding to scores of 1 and 2.

When it comes to comprehension scores, in Experiment 1, language is a significant factor for both high-performing and low-performing participants ($p = 0.01$ and $p < 0.01$, respectively). However, effect size is larger for low performers ($\eta^2 = 0.23$) than for high performers ($\eta^2 = 0.17$). In the case of high performers, both Epsilon ($p = 0.01$, $\rho = 0.18$) and VMTL ($p = 0.03$, $\rho = 0.16$) are associated with significantly higher scores compared to Henshin, while the difference between scores obtained under Epsilon and VMTL is not statistically significant ($p = 0.66$). Similar relative score differences can be observed for low-performing participants: Epsilon ($p < 0.01$, $\rho = -0.07$) and VMTL ($p = 0.04$, $\rho = -0.05$) are associated with significantly higher scores compared to Henshin, while the difference between scores obtained under Epsilon and VMTL is not statistically significant ($p = 0.16$). On the other hand, in Experiment 2, language only has a statistically significant effect on comprehension scores for low-performing participants ($p = 0.02$, $\eta^2 = 0.18$). For this participant group, VMTL is associated with significantly higher scores than Henshin ($p = 0.02$, $\rho = -0.26$), while other score differences are not statistically significant.

Completion times for the comprehension task are shown in the second group of data columns in Tables 7 and 8, respectively. They are illustrated as box plots in Fig. 10. A first observation is that completion times are longer for
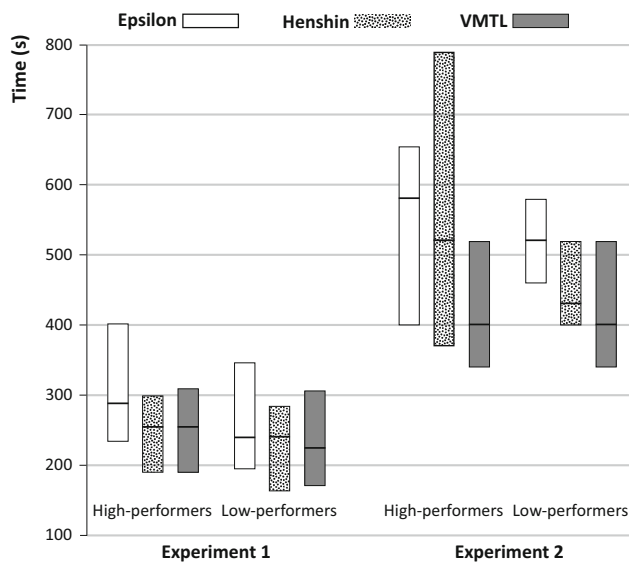
**Fig. 10** *Box plots* illustrating completion times of the comprehension questions for each MTL, grouped by experiment and participant performance



**Fig. 11** *Stacked bar* graphs illustrating cognitive load ratings for each MTL. *Lighter colors* correspond to higher ratings: *white* sections show the proportion of participants assigning the maximum rating (5), *dark grey* sections show the proportion of participants assigning the minimum rating (1) (color figure online)

Experiment 2, which features slightly more complex transformations. In terms of the effect of language, Experiment 1 participants required more time to answer questions under the Epsilon language than under the other two MTLs, although the difference is only slightly significant for high performers ($p = 0.06$, $\eta^2 = 0.12$), and not significant for low performers. A similar trend is visible for low performers in Experiment 2, but again lacking significance. In contrast, the completion times observed for high performers in Experiment 2 are highly dependent on language ($p < 0.01$, $\eta^2 = 0.2$). Here, VMTL is possibly associated with shorter completion times than both Epsilon ($p = 0.07$) and Henshin ($p = 0.02$, $\rho = 0.27$), while Epsilon is possibly associated with shorter completion times than Henshin ($p = 0.06$).

Difficulty and effort ratings are summarized in the rightmost data columns of Tables 7 and 8, and illustrated in Fig. 11 as stacked bar graphs. The bars in Fig. 11 are based on a 5-point scale of possible rating values. Lighter colors correspond to higher difficulty and effort ratings. In the case of Experiment 1, difficulty ratings do not significantly differ as a function of the considered transformation language. The only visually apparent difference, the higher difficulty ratings assigned by high-performing participants to VMTL, is not statistically significant ($p = 0.23$ and $p = 0.33$ when compared to Epsilon and Henshin, respectively). Similarly low differences in difficulty ratings can be observed in the case of Experiment 2. The only exception is represented by the potentially significantly higher difficulty ratings assigned by high-performing participants to Henshin compared to VMTL ($p = 0.05$, $\rho = -0.11$).
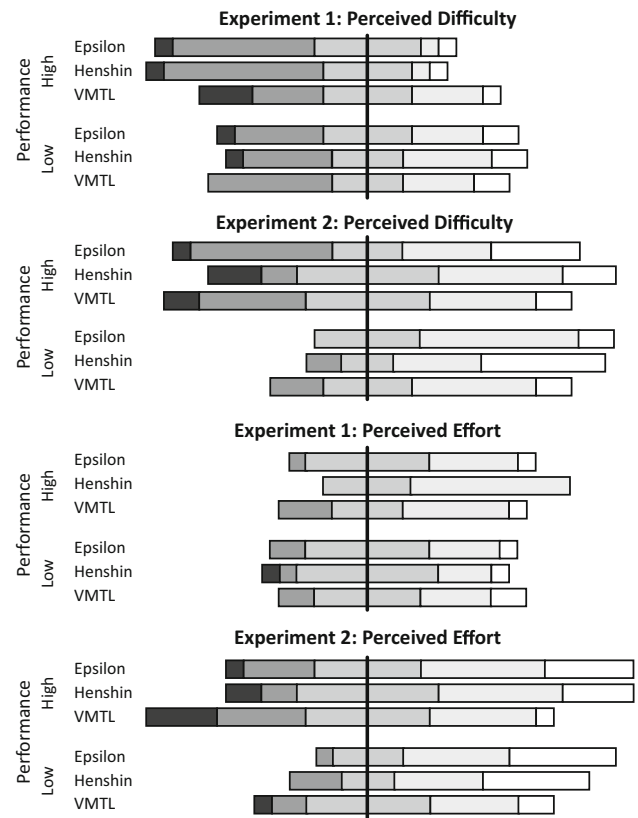
For Experiment 1, effort ratings generally follow the same pattern as difficulty ratings. However, whereas VMTL was rated as slightly more difficult by high-performing participants, the same participants appear to rate Henshin as requiring higher effort, though the increase is not statistically significant ($p = 0.75$ and $p = 0.44$ when compared to Epsilon and VMTL, respectively). The only statistically significant impact of an MTL on any of the cognitive load measurements is registered for the effort ratings of Experiment 2. Here, high-performing participants rate VMTL as requiring significantly less effort than Henshin ($p < 0.01$, $\rho = 0.41$), and potentially significantly less effort than Epsilon ($p = 0.1$, $\rho = 0.19$). A similar trend, but lacking statistical significance, can be observed for low-performing participants.

### 5.1.3 Interpretation

Our results show that VMTL outperforms Henshin in terms of comprehension scores. However, comprehension scores obtained under Epsilon are slightly higher than those obtained under VMTL. At the same time, VMTL is associ-
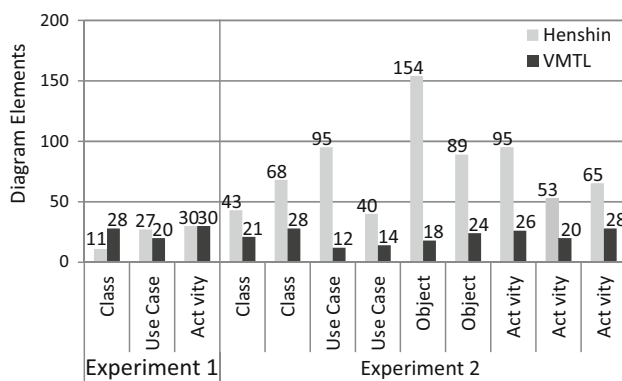
**Fig. 12** Number of diagram elements (*shapes*, *line segments*, labels) included in the Henshin and VMTL specifications presented to participants

ated with shorter completion times and lower cognitive load ratings than Epsilon.

The comprehension score differences between the visual MTLs (VMTL and Henshin) can be interpreted in at least two ways. First, the superiority of VMTL may result from its adoption of concrete syntax. The very reason why UML and other modeling languages adopt a concrete syntax on top of the abstract one is to improve comprehension. This is achieved by employing expressive visual notations and hiding unessential meta-model details. A second explanation for VMTL's better performance has to do with specification size. As shown in Fig. 12, the VMTL specifications included in Experiment 2 are considerably more concise than their Henshin counterparts. Completion times observed for Henshin in this experiment are significantly higher, as are cognitive load ratings (although Henshin obtains better comprehension scores for high-performing participants). The hypothesis that diagram size has an important effect on comprehension is supported by previous findings on UML diagram understanding [50].

The high comprehension scores associated with Epsilon are, to us, the most surprising result of the two experiments. Considered together with the high task completion times, Epsilon's high comprehension scores point to a higher level of engagement of participants with this language. We offer two possible explanations for this. The first is participant background, given that participants are Computer Science students with strong programming skills—over 80 % rated their own programming skills as good. With its C-style syntax and imperative execution semantics, Epsilon may have appeared familiar to them. This hypothesis suggests that repeating our experiments with actual end-user modelers as participants may yield a different outcome.

An alternative explanation for the high comprehension scores associated with Epsilon is offered by *cognitive fit theory* [61], which has primarily been applied in the field of information visualization [58,62]. According to this theory,

the accuracy of a problem solving process increases when the problem solving task matches the problem representation. In our case, the answer options of the comprehension questions are the problem solving task, and the transformation languages are the problem representation. Because the task is represented textually, a textual MTL such as Epsilon represents a better fit for solving it. The cognitive fit hypothesis is supported by a participant's remark in a follow-up interview: *"I couldn't relate the text to the pictures"*. Were this to be true, an experiment providing visual answer options may yield different outcomes.

The low completion times and cognitive load ratings observed for VMTL appear to suggest that its simple syntax has promoted an intuition-based approach to question answering. While fast and not very demanding, relying on intuition is not always accurate, as shown by VMTL's slightly lower comprehension scores compared to Epsilon.

Finally, the differences between the results obtained by participants in the two experiments are largely unsurprising. The increased difficulty of the questions in Experiment 2 are very likely the cause of the increases in completion times and cognitive load ratings observed for all three languages. The comprehension score differences are, however, more intriguing. First, question difficulty does not seem to have an important effect on the scores obtained by either high- or low-performing participants under VMTL and Henshin. This comes in contrast with Epsilon, under which low-performing participants score considerably worse when question difficulty increases. In this case, we hypothesize that the combination of high task difficulty and a relatively complex notation makes it unlikely for less skilled users to intuitively guess the intention of a transformation.

### 5.1.4 Threats to validity

To evaluate the validity of our experiments, we consider the four categories of threats to the validity of software engineering experiments described by Wohlin et al. [66].

*Construct validity* An experiment manifests construct validity if it measures the actual phenomena under investigation—in our case, model transformation language learnability. A possible threat to the construct validity of our experiments is the lack of a production task in which participants create transformation specifications themselves. While such an evaluation would be a desirable addition (especially for an evaluation of extended learnability), the comprehension tasks employed in our experiments offer an appropriate measure of initial learnability. Applying Cronbach's alpha to the comprehension resuts of Experiment 1,[9] however, reveals values below 0.5 for all languages. Although partially jus-

---

[9] Due to the adopted experimental design, Cronbach's alpha cannot be reliably computed for Experiment 2.

tified by the low number of questions per language, these values may indicate a low reliability of the questions. In what concerns cognitive load, it has been shown that the subjective measures employed in our experiments are highly correlated with objective cognitive load measures (cf. [18]). Last but not least, since VMTL is developed by the authors of this study, any bias in favor of this language must be avoided. To this end, we have presented the MTLs to participants in an impartial manner, replacing their names with pseudonyms.

*Internal validity* An experiment manifests internal validity if a causal conclusion regarding the phenomena under investigation can be drawn from it. The internal validity of our experiments is threatened by *learning effects*, a typical issue for within-subject experimental designs. Experiment 1 is particularly vulnerable, as it presents participants with the same questions for every MTL. To counter this threat, we have randomly assigned participants to one of three treatments, each presenting the MTLs in a different order. We have also investigated the statistical significance of language order. ANOVA yields $p = 0.02$ (i.e., language order is significant), while Fisher's exact test yields $p = 0.42$ (i.e., language order is not significant). Thus, the possibility that language order is significant cannot be ruled out. Learning effects are a much smaller threat for Experiment 2, as it does not reuse questions. However, Experiment 2 is under risk of confounding the effect of the MTLs with that of particular UML diagram types. For this reason, we have also created three versions of the questionnaire used in Experiment 2 by permuting the questions asked under each MTL. Finally, we have avoided selection bias (i.e., the self-selection of only those volunteers that are interested in the topic of the experiment) by offering a small participation prize.

*Conclusion validity* An experiment manifests conclusion validity if the statistical relationship between its factors and outcomes is correctly evaluated. Threats to conclusion validity typically originate in incomplete or incorrect statistical analysis procedures. We avoid such threats by presenting both descriptive and inferential statistics, verifying the assumptions of the employed statistical tests, performing nonparametric hypothesis testing (the Wilcoxon signed-rank test), and reporting effect sizes. In particular, the assumptions required by the ANOVA technique were verified by visually inspecting residual plots and Q-Q plots, as suggested by Montgomery et al. [33].

*External validity* An experiment manifests external validity if its outcomes can be generalized to a wider population. We ensure external validity by using sufficiently large numbers of participants. However, it may be argued that, as CS students, these participants are not representative for the population of end-user modelers. This is partially true, as over 80% of them have rated their own programming skills as good or very good, which cannot be said about the typical end-user modeler. However, a similar percentage of partic-

ipants have rated their knowledge of UML as good or very good, and their knowledge of OCL and MT as poor or very poor. These ratings are in line with the skills of end-user modelers illustrated in Table 1. Concerns can also be raised regarding the representatives of the transformations included in the experiments. To address them, we have included transformations on several diagram types covering both structural and behavioral models. Finally, the specifications used are relatively small (see Table 5; Fig. 12), a limitation difficult to avoid within the confines of an experiment.

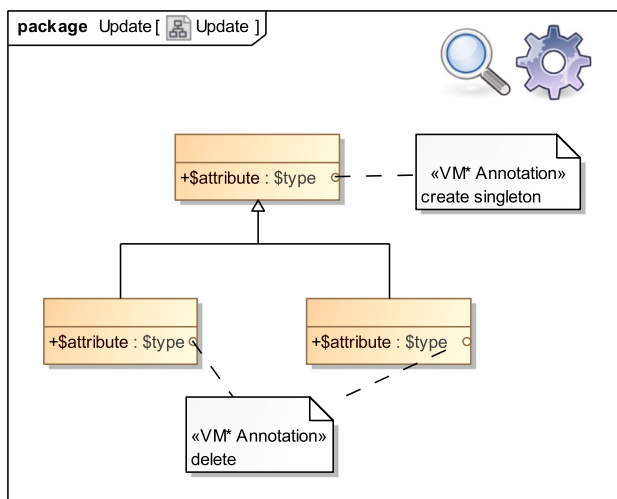## 5.2 Think-aloud protocol analysis

### 5.2.1 Methods and materials

While the experiments presented in Sect. 5.1 offer an objective measure of VMTL's relative learnability compared to that of Epsilon and Henshin, they do not provide any insights into how users approach the task of comprehending a VMTL specification. To gain these insights, we have conducted a think-aloud protocol investigation following the methodology described in [31]. Our study follows the *concurrent* think-aloud protocol methodology, in which participants verbally describe their thought process as they complete a given task.

Four PhD students with diverse backgrounds took part in the study. They are identified in what follows as Participant 1 through 4. The research domain, as well as the self-reported UML and English language skill levels of each participant are listed in Table 9. Notably, they all rate their UML knowledge as poor. Participants were presented with the three VMTL transformation specifications shown in Fig. 13. These transformations respectively operate on UML class diagrams (Transformation 1), UML activity diagrams (Transformation 2), and UML use case diagrams (Transformation 3).

After sitting through an introduction to MT and VMTL, participants were asked to read the VMTL specifications and explain their intended meaning. While doing so, they were allowed to consult a written specification of VMTL at any time. The experimenter took notes of participants' answers and other remarks as they completed the tasks, without providing assistance.

**Table 9** Backgrounds of think-aloud protocol analysis participants. UML and English language skills were self-assessed

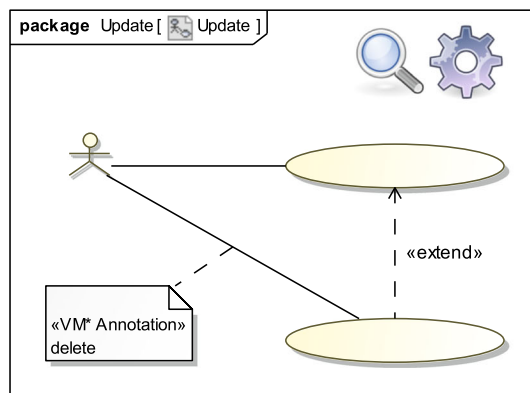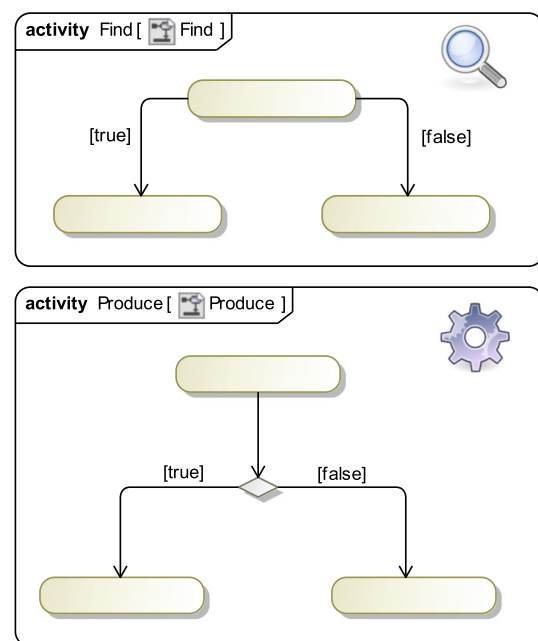| ID | Domain | UML [1..5] | English [1..5] |
|----|--------|------------|----------------|
| 1 | Nutrition science | 1 | 4 |
| 2 | Theoretical CS | 1 | 4 |
| 3 | SE | 2 | 2 |
| 4 | SE | 2 | 3 |

**Transformation 1:**

If two Classes have a common superclass, and they both have a public Attribute with the same name and type, delete this Attribute from both Classes and add it to the superclass if the superclass does not already own an Attribute with the same name and type.

**Transformation 2:**

If an Action has two outgoing Control Flows to other Actions guarded by "true" and "false", add a Decision Node between the Action and its outgoing Control Flows.

**Transformation 3:**

If two Use Cases are associated to the same Actor, and one of the Use Cases extends the other, delete the Association between the Actor and the extending Use Case.

**Fig. 13** VMTL transformation specifications included in the think-aloud protocol analysis. Transformation 1 is applicable to UML class diagram, Transformation 2 is applicable to UML activity diagrams, and Transformation 3 is applicable to UML use case diagrams

## 5.2.2 Results

The think-aloud protocol analysis has yielded several interesting findings. First, it has confirmed that allowing transformation developers to choose between `Update Patterns` and `Find`/`Replace Pattern` pairs is a good design decision. One participant found it difficult to relate a `Find Pattern` with its corresponding `Replace Pattern`, expressing a preference for the more concise notation of `Update Patterns`. Meanwhile, two other participants referred to `Find`/`Replace Pattern` pairs as *"before"* and *"after"* states of the transformation, a concept which they apparently found intuitive. Participants did not place any emphasis on pattern icons, indicating that offering these icons only as optional visual aids is indeed appropriate.

Additional observations of interest emerged regarding VMTL annotations. Two participants expressed confusion as to which pattern elements an annotation refers to in the context of a class diagram. Participant 4 could not precisely identify if the `delete` annotation in Transformation 1 refers to an Attribute or to its containing class, when the annotation in fact refers to the class. The same participant expressed surprise that an annotation can be anchored to an Association (as exemplified in Transformation 3), and could not determine which end of the Association it refers to—in fact, the annotation refers to the Association itself. Finally, Participant 2 repeatedly referred to the annotations as *"log messages"*. These observations suggest that VMTL's textual annotations may not be as intuitive as hoped, and that ambiguities introduced by the model editor regarding the anchor point of an annotation may negatively affect comprehension.

The now deprecated `create singleton` clause included in the version of VMTL used in the study has also caused participants some difficulty. This clause is employed in Transformation 1. All participants needed to consult its written description several times, and Participant 2 described it as *"taking an existing model element and turning it into a singleton"*. The clause's intended effect is in fact to create a new model element only if an identical one does not already exist in the source model. We suspect that the term "singleton" may have a different or unclear meaning to end-user modelers, and have consequently replaced the `create singleton` clause with the more explicit formulation of `create if not exists`.

Participants' individual backgrounds may have affected the way they approached the tasks. Participants with a CS background attempted to interpret the specifications by considering similar notions from general-purpose programming languages. For example, Participant 4 correctly remarked that VMTL's notation for variables resembles the notation adopted by the PHP scripting language. However, Participant 2's intuition to equate VMTL annotations with log messages was less helpful. As expected based on her non-computer science background, Participant 1 encountered difficulties in understanding the meaning and purpose of the various UML notations, with class diagrams posing the greatest challenge.

Ultimately, the limited scale of this think-aloud protocol analysis prevents us from attempting to generalize its outcomes. Nevertheless, the study has provided valuable indications regarding which aspects of VMTL require improvements. These indications are reflected in the current version of VMTL's syntax presented in Sect. 3.3. The analysis has also suggested possible contributing factors to the results obtained by VMTL in the learnability experiments presented in Sect. 5.1.

## 5.2.3 Threats to validity

The most important threats to the validity of this study are related to its participants. First, the low number of participants does not allow us to attempt a generalization of the outcomes to any wider population. The diverse backgrounds of participants also hinder any attempts at generalization, while their overall poor knowledge of UML arguably makes participants unrepresentative for the general population of end-user modelers. Under these circumstances, the qualitative data gathered from this study may only be used to complement the interpretation of our experimental results and inform some relatively minor updates to VMTL.

Participant selection bias is also a possible threat to validity, as all participants volunteered to take part in the study. This threat was mitigated by offering a small participation prize.

The instrumentation of the study poses a threat to its validity, as the three transformations employed only cover a small subset of UML. However, three different UML diagram types were included, featuring both structural and behavioral constructs. A similar observation can be made regarding the relatively small number of VMTL language constructs present in these transformations. While `Find`, `Produce`, and `Update` patterns were all included, pattern types representing application conditions were not included, and neither were most of VMTL's textual annotation clauses. Nevertheless, many of the observations presented in Sect. 5.2.2 are not limited to specific syntax elements, but are more generally applicable to entire categories of VMTL constructs, such as annotations or icons.

Finally, experimenter bias must be considered, since participants' qualitative feedback may mirror any apparent expectations on the experimenter's side. Participants in our study were aware of the fact that VMTL is a language developed by the experimenter. They were, however, not informed about the precise purpose of the study or any expectations regarding its outcomes.

# 6 Related work

## 6.1 Support for end-user modelers

The need for languages and tools supporting end-user modelers has first been identified in connection to model querying. The query-by-example approach adopted by Constraint Diagrams (CD [26]) and Join Point Designation Diagram (JPDD [46]) allows users to express queries as concrete syntax patterns. The same technique can be applied to specify model constraints [52]. The business process modeling community particularly emphasizes end-user friendly model querying through languages like BP-QL [11] and BPMN-Q [7].

VMTL itself has been developed on the foundation of VMQL [51], a usability-focused query-by-example language. Like VMTL, VMQL has been experimentally evaluated. In particular, it has been shown to offer superior usability compared to OCL, the de facto standard model query language, when querying UML models [51] and BPMN models [48].

## 6.2 Transparent model transformation

The insight that model transformations can be viewed as models [12] has been the first step toward syntax transparency. Early MTLs such as VMT [43] and MOLA [25] attempt to capture the usability benefits of syntax transparency by integrating concrete syntax fragments in their specification languages. More recent tools, such as the Web-based AToMPM [54], take a similar route. However, these approaches fail to achieve syntax transparency, as they modify their host languages' existing concrete syntax. The purpose of the modifications is typically to provide expressive rule execution control mechanisms [25,54].

A number of existing MTLs do, however, adhere to a subset of the transparency principles defined in Sect. 3.1—albeit without explicitly claiming this. These MTLs are listed in Table 10.

Syntax transparency is exhibited by MATA [64], PICS [8] and the MTL proposed by Schmidt [41], all of which deliberately shun expressive control flow in favor of avoiding conformance breaking augmentations to the host metamodel. However, none of these approaches also address environment or execution transparency. MATA can only be used via the IBM Rational Software Modeler, thus failing to provide environment transparency. It generates executable rules for the AGG [57] graph transformation engine. PICS, on the other hand, has never been implemented, and is designed to act exclusively as a front-end for graph transformation. The purely conceptual proposal by Schmidt is limited to endogenous UML transformations, and does not discuss model editor integration.

**Table 10** Approaches supporting explicit MT specifications and exhibiting syntax transparency (ST), environment transparency (EnT), or execution transparency (ExT)

| Approach | ST | EnT | ExT |
|---|---|---|---|
| AToMPM [54] | × | × | ✓ |
| MATA [64] | ✓ | × | ✓ |
| MeTAGeM [14] | × | × | ✓ |
| MoTif [56] | × | × | ✓ |
| MoTMoT [59] | × | ✓ | ✓ |
| PICS [8] | ✓ | × | × |
| QVT-R [36] | × | × | ✓ |
| Schmidt [41] | ✓ | × | × |
| TransML [21] | × | × | ✓ |
| VMTL | ✓ | ✓ | ✓ |

Model transformation literature typically only mentions environment transparency as a consequence of syntax transparency. One of the few transformation solutions exhibiting environment transparency in the absence of syntax transparency is MoTMoT [59], which defines a UML profile allowing the specification of graph transformations using any UML editor. MoTMoT specifications can be executed by any graph transformation engine, thus also exhibiting execution transparency.

Several high-level MTLs are implemented by translation to lower level languages, in effect achieving execution transparency. The QVT Relations [36] standard, meant to be implemented by translation to QVT Core, is one such example. The AToMPM tool mentioned above generates executable specifications for a transformation engine based on the Python programming language. The recent emergence of transformation primitive libraries such as T-Core [55], along with their usage for implementing existing MTLs such as MoTif [56], indicates that this implementation style is viable. Execution transparency is also addressed in the context of the systematic development of model transformations by the transML [21] and MeTAGeM [14] tools.

While VMTL shares some commonalities with each of these MTLs, it is the first model transformation approach to encompass syntax, environment, and execution transparency. Its focus on end-user modelers also sets VMTL apart, as the majority of the MTLs in Table 10 were proposed in an MDE context.

Apart from the approaches listed in Table 10, Model Transformation By-Example (MTBE [60]) is a paradigm aimed at enabling modelers to express transformations while largely circumventing the use of dedicated MTLs. In MTBE, transformations are defined as concrete syntax examples from which an underlying engine deduces rules expressed using a traditional MTL. In *correspondence-based* approaches [6,9,27,39,65], users explicitly state the cor-

respondences between example source and target model elements. In *demonstration-based* approaches [30,53], transformation rules are exemplified by performing edit operations on the source model. Both MTBE styles exhibit syntax and execution transparency. However, there is an important difference between MTBE approaches and MTLs supporting explicit transformation specifications, such as VMTL. Whereas in MTBE transformation rules are *deduced* from concrete syntax examples, the MTLs listed in Table 10 support *specifying* these rules directly.

### 6.3 Usability in model transformation

When discussing model transformation approaches, usability is often claimed but rarely investigated. The application of sound empirical methods to this research field appears to be in its infancy.

Most existing empirical evidence regarding MTL usability is of a qualitative nature. Silva et al. [45] investigate the extent to which a selection of eight MTLs cater to the needs of novice users, finding that most do not adequately support this user category. The study is based exclusively on its authors' evaluation of a set of features deemed desirable for novices. Neither the selection of these features nor the choice of evaluated MTLs is discussed, and the topic of language learnability is not addressed. Grønmo et al. [19] investigate the usability of three model transformation languages by comparing their conciseness and required development effort for implementing a complex model transformation. The choice of MTLs included in this study is similar to ours: a textual MTL, an abstract syntax visual MTL, and a concrete syntax visual MTL. However, the presented evaluation is limited to an intuition-guided discussion of the three transformation specifications.

Two user studies evaluating the usability of model transformation approaches have been reported in the literature: the study by Avazpour et al. [6] and the one by Batory et al. [10]. The first study evaluates the CONVErT framework, a Model Transformation By-Example approach enhanced with interactive recommendations. The presented evaluation is primarily qualitative, and features a total of 15 participants. Each participant was asked to complete a transformation task using CONVErT, and subsequently provide a subjective account of the experience via a questionnaire. Most participants rated the framework as easy to learn and use, but data describing their objective performance is not provided. No other MTLs were included in the evaluation.

Batory et al. [10] adopt learnability as the main design goal of their model transformation approach, MDELite. Following a failed attempt to teach MT using EMF, the authors state that the EMF toolchain promotes *"incantations to solve problems"*, and consequently propose MDELite, a library for the Prolog logic programming language. The usability of MDELite is compared to that of the Atlas Transformation Language (ATL, [24]) in a quasi-experiment including 12 undergraduate and graduate CS students as participants. The main findings of this study are that MDELite does not objectively improve students' productivity, but is nevertheless perceived by students as significantly easier to learn and use than ATL. The presented evaluation is rich in qualitative information, especially participant remarks, but provides little quantitative evidence. This comes in contrast with the primarily quantitative evaluation of VMTL presented in Sect. 5.1.

## 7 Conclusions

### 7.1 Summary

End-user modelers can broadly be described as highly qualified but possibly non-technical domain experts creating and using models as part of their work. They are expected to be closely familiar with some particular modeling languages, but have no incentive to master meta-modeling or rule languages such as OCL. Furthermore, the majority of end-user modelers are not software developers, instead fulfilling roles such as "business analyst" or "enterprise architect."

Many of the tasks involved in the life-cycle of a model, such as quality-oriented refactoring or migration to an updated meta-model, can fall within the responsibility of end-user modelers. Considering that these tasks can be seen as model transformations, end-user modelers are currently at a technological disadvantage compared to MDE practitioners. This latter group of users has access to a large variety of MTLs, almost all designed to accommodate their existing software engineering skills. End-user modelers are left unable to use, and often unaware of MT technologies that could greatly benefit their productivity.

We have addressed this problem by systematically developing an MTL for end-user modelers. First, we have investigated the requirements for such an MTL, and synthesized them into the general concept of transparent model transformation. This concept stands on three pillars: syntax, environment, and execution transparency. Together, these principles ensure that end-user modelers can access up-to-date MT technology using exclusively languages and tools they are familiar with, requiring minimal or no extensions.

We have then defined VMTL, the first MTL to respect all three transparency principles. VMTL maps the full range of constructs typically found in a declarative transformation language to elements of the host modeling language. This process amounts to a lightweight meta-model extension that does not break compatibility with existing model editors. Transformation specifications created this way can then be executed by any sufficiently expressive transforma-

tion engine. In our implementation, we have used the Henshin transformation engine, as it offers an operational semantics closely resembling that of VMTL. The need for extending non-MDE model editors to support VMTL is mitigated by the option of deploying VMTL's entire runtime infrastructure as a RESTful web service API.

The argument in favor of VMTL is one of superior learnability, an argument requiring empirical confirmation. We have therefore conducted complementary empirical investigations into VMTL's learnability, yielding both quantitative and qualitative evidence.

We first performed two user experiments comparing VMTL with a textual MTL (Epsilon) and an abstract syntax visual MTL (Henshin) from a learnability standpoint. Our evaluation was based on two task metrics (comprehension score and task completion time) and two subjective metrics measuring the cognitive load imposed by each language on participants (perceived difficulty and effort). VMTL was associated with the shortest completion times and lowest cognitive load ratings, but also with comprehension scores slightly below Epsilon. We hypothesize that VMTL outperformed Henshin either due to its use of concrete syntax, or due to the known effect of diagram size on comprehension. We also hypothesize that the cognitive fit between Epsilon, a textual language, and the textual questions included in the experiment may have benefited this MTL.

To understand how users approach VMTL specifications, we have also carried out a concurrent think-aloud protocol analysis. This has confirmed some of the design decisions adopted for VMTL, and has also yielded a list of possible syntax improvements.

### 7.2 Contributions

We have introduced the notion of end-user modeler, and characterized end-user modelers in contrast to MDE practitioners. Based on the principles of transparent model transformation, which we have first defined in a previous publication [2], we have proposed VMTL as the first model transformation language explicitly addressing the needs and capabilities of end-user modelers. We have provided detailed descriptions of VMTL's syntax, operational semantics, and implementation. Finally, we have presented and discussed the results of two experiments addressing VMTL's learnability, as well as those of a think-aloud protocol analysis aimed at uncovering its potential shortcomings.

We believe that both researchers and practitioners can benefit from these contributions. From a research perspective, the emphasis placed in this paper (and recent similar publications [6,10]) on the empirical validation of usability claims illustrates the importance of human factors studies in the area of model transformation languages. The existence of such studies constitutes a strong argument in favor of the

industrial adoption of these languages in both MDE and end-user modeler contexts. At the same time, this paper's focus on end-user modelers can help bring a significant new category of modeling practitioners to the attention of the model transformation research community.

### 7.3 Future work

We intend to continuously improve VMTL based on empirical results. Such improvements are best informed by qualitative evidence, motivating us to emphasize interviews and think-aloud protocol studies in the future. The results of the presented think-aloud protocol analysis also encourage us to pursue this direction.
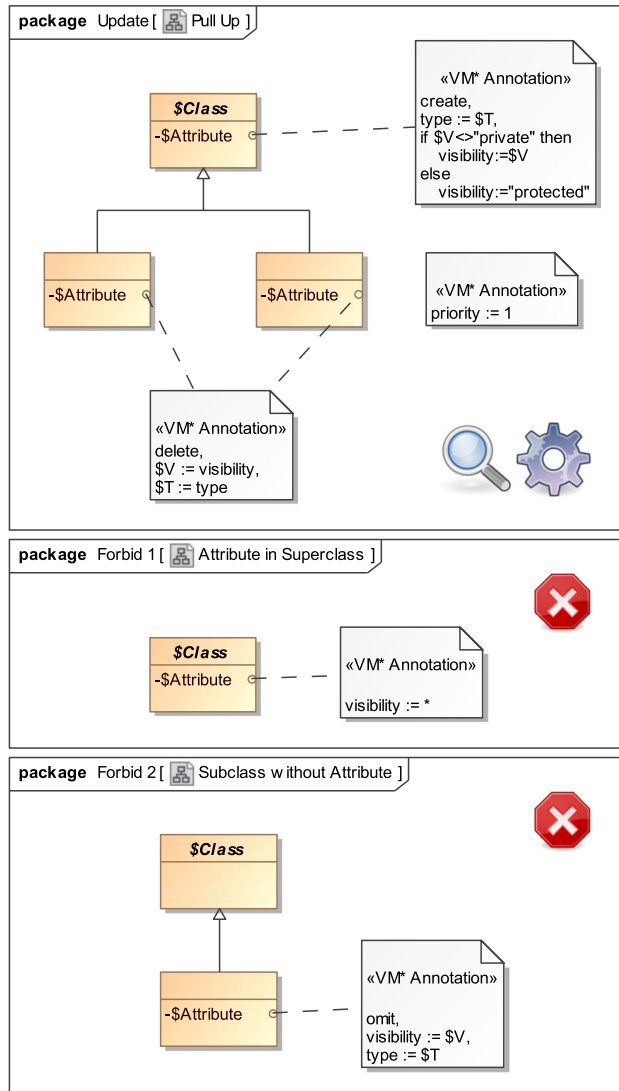
At the same time, we plan to perform a number of follow-up experiments on MTL learnability. First, including participants with a background more closely resembling that of end-user modelers will improve the validity of our results. The inclusion of participants without a computer programming background will also help eliminate the possible bias in favor of textual MTLs such as Epsilon. The hypothesized cognitive fit advantage enjoyed by Epsilon as a textual language could be mitigated by providing visual answer options for the comprehension questions. Studying additional MTLs would allow us to either more confidently generalize or re-consider our conclusions. Finally, asking participants to produce transformation specifications themselves, as opposed to only comprehending existing specifications, is an advisable variation to our experimental design.

Extending VMTL's tool support is an equally important future work direction. A particularly relevant concern in this direction is the provision of end-user accessible support for debugging VMTL specifications. Furthermore, since VMTL specifications can currently only be executed using the Henshin transformation engine, extending the VM* Runtime to accommodate alternative engines will lend credibility to VMTL's syntax transparency claims. We also intend to leverage VMTL's service-based deployment to create lightweight model transformation plugins for traditional model editors such as Microsoft Visio and MagicDraw.

## Appendix: The pull-up attribute refactoring in VMTL

As a somewhat more elaborate VMTL specification compared to the introductory examples featured in Sect. 2, this appendix presents the VMTL specification of the "Pull-Up Attribute" refactoring. This refactoring addresses a consecrated UML class diagram design anti-pattern [1]. Its description states that common attributes of all classes sharing the same abstract superclass must be deleted, and an
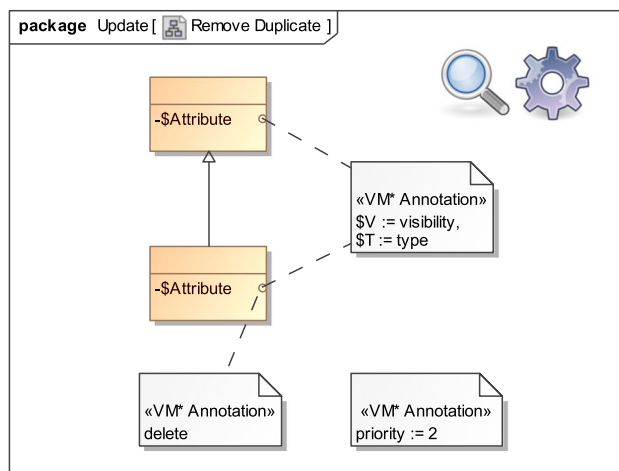
*Rule 1*



*Rule 2*



**Fig. 14** A multi-rule VMTL specification: the "Pull-Up Attribute" refactoring. Rule 1 (*top*) and Rule 2 (*bottom*)

attribute with the same `name`, `type`, and `visibility` (i.e., the same *signature*) must be created in the superclass. In 1, this is the case for the "amount" and "confidential notes" attributes shared by the "Installment Loan" and "Revolving Loan" classes. The VMTL implementation of this refactoring relies on two rules. Rule 1, shown in Fig. 14 (top) addresses the basic case with only two subclasses, while Rule 2, shown in Fig. 14 (bottom), handles additional subclasses.

Rule 1 consists of an `Update Pattern` and two `Forbid Patterns`. The `Update Pattern`, named "Pull Up," will match any class that has at least two subclasses sharing an attribute with the same signature. The `name` of the class is stored in the `$Class` variable, while the `name`, `visibility`, and `type` of the attribute are stored in the `$Attribute`, `$V`, and `$T` variables, respectively. The `delete` annotation is used to remove the attribute from the subclasses, while the `create` annotation creates a new attribute in the superclass. The `name`, `type`, and `visibility` of the new attribute are set to the values stored in the `$Attribute`, `$V`, and `$T` variables. Using VMTL's `if` operator, the `visibility` of the new attribute is set to `protected` if the deleted attribute's visibility was `private`, so that it is visible to subclasses.

The two `Forbid Patterns` of Rule 1 act as application conditions. If any one of them is matched, the rule will no longer be applied to that specific source model fragment, regardless if the `Update Pattern` is matched. The first `Forbid Pattern`, named "Attribute in Superclass", ensures that the rule is not applied if the attribute to be pulled up already exists in the superclass. The `visibility := *` annotation allows the pattern to match any attribute visibility value. Finally, the refactoring should only be applied if *all* subclasses of the considered class own the attribute to be pulled up. This condition is enforced by the "Subclass without Attribute" `Forbid Pattern` using the `omit` annotation. Whenever the `omit` annotation is anchored to a pattern element, that element must not appear in a successful pattern match.

Rule 2 addresses the scenario in which there are more than two subclasses owning an attribute to be pulled up. Since an identical attribute has already been created in the superclass, this rule removes all attributes appearing in both the superclass and its subclasses. To this end, a single `Update Pattern` with no additional application conditions is required.

## References

1. Arendt, T.: Quality Assurance of Software Models. Ph.D. thesis, Philipps-Universität Marburg (2014)
2. Acretoaie, V., Störrle, H., Strüber, D.: Transparent model transformation: turning your favourite model editor into a transformation

tool. In: Theory and Practice of Model Transformation, LNCS, vol. 9152, pp. 121–130. Springer, Berlin (2015)

3. Acretoaie, V., Störrle, H.: Hypersonic: Model Analysis and Checking in the Cloud. In: Proceedings of 2nd Workshop on Scalability in Model Driven Engineering. CEUR Workshop Proceedings, vol. 1206, pp. 6–13 (2014)

4. Acretoaie, V., Störrle, H.: MQ-2: a tool for prolog-based model querying. In: Joint Proceedings of Co-located Events at the 8th European Conference on Modelling Foundations and Applications (ECMFA'12), pp. 328–331. Technical University of Denmark, Kgs. Lyngby, DK (2012)

5. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. Model Driven Engineering Languages and Systems. LNCS, vpl. 6394, pp. 121–135. Springer, Berlin (2010)

6. Avazpour, I., Grundy, J., Grunske, L.: Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations. J. Visual Lang. Comput. **28**, 195–211 (2015)

7. Awad, A., Sakr, S.: On efficient processing of BPMN-Q queries. Comput. Ind. **63**(9), 867–881 (2012)

8. Baar, T., Whittle, J.: On the usage of concrete syntax in model transformation rules. In: Perspectives of Systems Informatics. LNCS, vol. 4378, pp. 84–97. Springer, Berlin (2007)

9. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Softw. Syst. Model. **8**(3), 347–364 (2009)

10. Batory, D., Azanza, M.: Teaching model-driven engineering from a relational database perspective. Softw. Syst. Model. pp. 1–25 (2015)

11. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes with BP-QL. Inf. Syst. **33**(6), 477–507 (2008)

12. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? Transformation models! In: Model-driven engineering languages and systems, LNCS, vol. 4199, pp. 440–453. Springer, Berlin (2006)

13. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. Softw. Syst. Model. **11**(2), 227–250 (2012)

14. Bollati, V.A., Vara, J.M., Jiménez, Á., Marcos, E.: Applying MDE to the (semi-)automatic development of model transformations. Inf. Softw. Technol. **55**(4), 699–718 (2013)

15. Cohen, J.: Statistical Power Analysis for the Behavioral Sciences, 2 edn. Routledge (1988)

16. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)

17. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Berlin (2006)

18. Gopher, D., Braune, R.: On the psychophysics of workload: Why bother with subjective measures? Hum. Factors **26**(5), 519–532 (1984)

19. Grønmo, R., Møller-Pedersen, B., Olsen, G.K.: Comparison of Three Model Transformation Languages. Model Driven Architecture—Foundations and Applications. LNCS, **5562**, pp. 2–17. Springer, Berlin (2009)

20. Grossman, T., Fitzmaurice, G., Attar, R.: A survey of software learnability: metrics, methodologies and guidelines. In: Proceedings of SIGCHI Conference on Human Factors in Computing Systems (CHI'09), pp. 649–658. ACM, New York (2009)

21. 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)

22. Internet Engineering Task Force (IETF): IETF RFC 6749: The OAuth 2.0 Authorization Framework. http://tools.ietf.org/html/rfc6749 (2012)

23. Jones, B., Kenward, M.G.: Design and Analysis of Cross-Over Trials, third edn. Chapman and Hall/CRC, London (2014)

24. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)

25. Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. In: Model Driven Architecture. LNCS, vol. 3599, pp. 62–76. Springer, Berlin (2005)

26. Kent, S.: Constraint diagrams: visualizing invariants in object-oriented models. In: Proceedings of 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97), pp. 327–341. ACM (1997)

27. Kessentini, M., Sahraoui, H., Boukadoum, M., Omar, O.B.: Search-based model transformation by example. Softw. Syst. Model. **11**(2), 209–226 (2012)

28. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon object language (EOL). In: Model Driven Architecture—Foundations and Applications. LNCS, vol. 4066, pp. 128–142. Springer, Berlin (2006)

29. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit transformation modeling. In: Models in Software Engineering, LNCS, vol. 6002, pp. 240–255. Springer, Berlin (2010)

30. Langer, P., Wimmer, M., Kappel, G.: Model-to-model transformations by demonstration. In: Theory and Practice of Model Transformations. LNCS, vol. 6142, pp. 153–167. Springer, Berlin (2010)

31. Lewis, C.: Using the "Thinking Aloud" Method in Cognitive Interface Design. Tech. Rep. RC-9265, IBM (1982)

32. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)

33. Montgomery, D.C.: Design and Analysis of Experiments, 8th edn. Wiley, London (2012)

34. Obeo: EMF Compare. https://www.eclipse.org/emf/compare/

35. Object Management Group: Business Process Model and Notation (BPMN) 2.0.2. OMG Document formal/2013-12-09 (2013)

36. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification 1.2. OMG Document formal/15-02-01 (2015)

37. Object Management Group: Object Constraint Language 2.4. OMG Document formal/2014-02-03 (2014)

38. Object Management Group: Unified Modeling Language (UML) 2.5. OMG Document formal/2015-03-01 (2015)

39. Saada, H., Dolques, X., Huchard, M., Nebut, C., Sahraoui, H.: Generation of operational transformation rules from examples of model transformations. In: Model Driven Engineering Languages and Systems. LNCS, vol. 7590, pp. 546–561. Springer, Berlin (2012)

40. Sarbanes-Oxley Act: 107th Congress Public Law 204. US Government Printing Office (2002)

41. Schmidt, M.: Transformations of UML 2 models using concrete syntax patterns. In: Rapid Integration of Software Engineering Techniques. LNCS, vol. 4401, pp. 130–143. Springer, Berlin (2007)

42. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003)

43. Sendall, S., Perrouin, G., Guelfi, N., Biberstein, O.: Supporting Model-to-Model Transformations: The VMT Approach. Tech. Rep. LGL-REPORT-2003-005, École Polytechnique Fédérale de Lausanne (2003)

44. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)

45. Silva, G.C., Rose, L.M., Calinescu, R.: A qualitative study of model transformation development approaches: supporting novice developers 18–27. In: Proceedings of International Workshop on Model-Driven Development Processes and Practices (MD2P2'14), CEUR Workshop Proceedings, vol. 1249, pp. 18–27 (2014)

46. Stein, D., Hanenberg, S., Unland, R.: Join point designation diagrams: a graphical representation of join point selections. Int. J. Softw. Eng. Knowl. Eng. **16**(3), 317–346 (2006)

47. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, second edn. Addison-Wesley, Reading (2008)

48. Störrle, H., Acretoaie, V.: Querying business process models with VMQL. In: Proceedings of ACM SIGCHI Annual International Workshop on Behaviour Modelling—Foundations and Applications (BMFA'13), pp. 41–410. ACM, New York (2013)

49. Störrle, H.: MOCQL: a declarative language for ad-hoc model querying. In: Modelling Foundations and Applications. LNCS, vol. 7949, pp. 3–19. Springer, Berlin (2013)

50. Störrle, H.: On the impact of layout quality to understanding UML diagrams: size matters. In: Model-Driven Engineering Languages and Systems, LNCS, vol. 8767, pp. 518–534. Springer, Berlin (2014)

51. Störrle, H.: VMQL: A Visual Language for Ad-Hoc Model Querying. J. Visual Lang. Comput. **22**(1) (2011)

52. Stricker, V., Hanenberg, S., Stein, D.: Designing design constraints in the UML using join point designation diagrams. In: Objects, Components, Models and Patterns, LNBIP, vol. 33, pp. 57–76. Springer, Berlin (2009)

53. Sun, Y., White, J., Gray, J.: Model Transformation by Demonstration. Model Driven Engineering Languages and Systems. LNCS, vol. 5795, pp. 712–726. Springer, Berlin (2009)

54. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Huseyin, E.: AToMPM: a web-based modeling environment. In: Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition, CEUR Workshop Proceedings, vol. 1115, pp. 21–25 (2013)

55. Syriani, E., Vangheluwe, H., LaShomb, B.: T-core: a framework for custom-built model transformation engines. Softw. Syst. Model. **13**(3), 1–29 (2013)

56. Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. Softw. Syst. Model. **12**(2), 387–414 (2011)

57. Taentzer, G.: AGG: a graph transformation environment for modeling and validation of software. In: Applications of Graph Transformations with Industrial Relevance. LNCS, vol. 3062, pp. 446–453. Springer, Berlin (2004)

58. Teets, J., Tegarden, D., Russell, R.: Using cognitive fit theory to evaluate the effectiveness of information visualizations: an example using quality assurance data. IEEE Trans. Vis. Comput. Gr. **16**(5), 841–853 (2010)

59. Van Gorp, P., Keller, A., Janssens, D.: Transformation language integration based on profiles and higher order transformations. In: Software Language Engineering. LNCS, vol. 5452, pp. 208–226. Springer, Berlin (2009)

60. Varró, D.: Model transformation by example. In: Model Driven Engineering Languages and Systems. LNCS, vol. 4199, pp. 410–424. Springer, Berlin (2006)

61. Vessey, I.: The theory of cognitive fit: one aspect of a general theory of problem-solving? In: Human-Computer Interaction and Management Information Systems, pp. 141–183. Routledge (2006)

62. Vessey, I.: Cognitive fit: a theory-based analysis of the graphs versus tables literature. Decis. Sci. **22**(2), 219–240 (1991)

63. VMTL Experimental Replication Package. https://vmstar.compute.dtu.dk/doku.php?id=vmtl:evaluation

64. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A.: Arajo, J.: MATA: a unified approach for composing UML aspect models based on graph transformation. In: Transactions on Aspect-Oriented Software Development VI. LNCS, vol. 5560, pp. 191–237. Springer, Berlin (2009)

65. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: Proceedings of 40th Annual Hawaii International Conference on System Sciences, HICSS'07, p. 285b. IEEE Computer Society, Washington, DC (2007)

66. Wohlin, C., Runeson, P., Höst, M., C. Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Berlin (2012)

**Vlad Acreţoaie** received an M.Sc. in Computer Science from the Technical University of Denmark in 2012, and completed his Ph.D. studies at the same institution in 2016. His doctoral dissertation provides solutions for the adoption of model transformation, query, and constraint languages and tools by end-user modelers. His research interests are at the intersection of Model-Based and Empirical Software Engineering. Vlad currently works as a software developer in industry, where his primary focus is on Model-Driven Engineering.



**Harald Störrle** received a Dipl.-Inform. and a Dr. rer. nat. from the Universities of Hamburg (1997) and Munich (2000), respectively. From 2001 to 2009 he worked as a software architect and methodology consultant in industry, sidelining as an adjunct lecturer at the University of Munich. Starting 2006, he held lecturer positions at the Universities of Innsbruck and Munich. Since 2009 he is Associate Professor of Software Engineering at the Technical University of Denmark (DTU) in Lyngby near Copenhagen. He is a Senior Member of the ACM and elected member of the ACM Europe Council.



**Daniel Strüber** is a postdoctoral researcher at University of Marburg. In his doctoral thesis, Daniel investigated refactorings for large models and model transformations, key artifacts in the model-driven development of complex software systems. His research interests include strategies to control the variability in model-driven engineering and to enable flexible modeling. Daniel is an Eclipse committer in the EMF Henshin project.