VMTL: a language for end-user model transformation

Acretoaie, Vlad; Störrle, Harald; Strüber, Daniel

Published in:
Software and Systems Modeling

Link to article, DOI:
10.1007/s10270-016-0546-9

Publication date:
2016

Document Version
Peer reviewed version

Softw. Syst. Model. manuscript No.
(will be inserted by the editor)

VMTL: a language for end-user model transformation

V. Acreţoaie · H. Störrle

Received: date / Accepted: date

Abstract *Context:* Model transformation is closely associated with Model-Driven Engineering (MDE), and existing model transformation languages (MTLs) are shaped by and for MDE practitioners – a user group with specific needs and capabilities which are not characteristic of modelers in general. The current generation of MTLs are thus ill-equipped to address the requirements of end-user modelers (EUMs) in areas such as requirements engineering, business process management, enterprise architecture, or mechanical engineering.

Objective: We aim to characterize EUMs, informing the design and implementation of a novel MTL that is finely tuned to their skills and requirements. With this contribution, we hope to broaden the application scope of model transformation and MDE technology in general.

Method: We discuss the profile of EUMs and derive specific requirements. Based on these requirements, we identify the design principles behind MTLs for EUMs. We also propose the Visual Model Transformation Language (VMTL) as the first transformation language for EUMs. VMTL is based on the Visual Model Query Language (VMQL), which has already demonstrated its usability in the context of EUMs. We implement VMTL and empirically investigate its learnability via two experiments and one think-aloud protocol study.

Results: We have defined the principles of syntax transparency, environment transparency, and execution transparency. We have also provided an in-depth description of VMTL and discussed its conformance to these principles. VMTL's implementation based on the Henshin transformation engine demonstrates the language's feasibility. The results of our empirical evaluation indicate that VMTL compares favorably in terms of learnability with two state-of the-art MTLs: Epsilon and Henshin.

Keywords End-user modelers · Transparent Model Transformation · VMTL · Henshin · Epsilon · Learnability · Experiment · Think-aloud protocol

V. Acreţoaie · H. Störrle
Department of Applied Mathematics and Computer Science, Technical University of Denmark, Kgs. Lyngby, Denmark
E-mail: rvac@dtu.dk

H. Störrle
E-mail: hsto@dtu.dk

1 Introduction

Model transformation (MT) is "the heart and soul" [41] of Model-Driven Engineering (MDE), a software development paradigm in which models replace code as the central artifact of the software development process [40]. 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 [41]. 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 classic engineering disciplines. All such models are occasionally refactored, translated, and migrated. These MT application scenarios are currently unexploited due to the lack of suitable MTLs and supporting tools.

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

	Domain knowlege	**Modeling**	**Metamodeling**	**Rule languages**	**MT**	**Programming**
End-user modeler	✔	✔	✘	✘	✘	✘
MDE practitioner	✘	✔	✔	✔	✔	✔

1.1 End-user modelers

End-user modelers (EUMs) are highly trained domain experts, but do not have the level of Software Engineering expertise demanded to master today's MTLs. Their skills typically do not include rule languages such as the Object Constraint Language (OCL [35]), metamodeling, or computer programming. 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 any suitable MTL must be gentle, suggesting that MTLs should adopt modeling notations already in place in a given domain. This, however, entails that any MTL for EUMs must be sufficiently generic to be applicable in the context of very diverse modeling languages.

The second key concern for EUMs 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, is no longer relevant. Instead, various use cases involving consistent global model updates take center stage.

1.2 Transparent Model Transformation

To address the needs of EUMs, the model transformation must be *transparent*: it must focus on what the EUM wants to achieve rather than on technical aspects. In [4] we contend that an MTL for EUMs must exhibit syntax, environment, and execution transparency.

Syntax transparency refers to an MTLs ability to leverage the syntax of any host modeling language to specify transformations. A transformation specified using a syntax transparent MTL is simultaneously a valid model fragment in its host modeling language. EUMs benefit from this by not having to learn a new language for the sole purpose of specifying transformations.

Environment transparency indicates that an MTL does not impose constraints on the editor used to specify transformations. It comes as a direct consequence of syntax transparency, but can also exist separately. For instance, most existing textual MTLs are syntax transparent, as they can be used with any text editor. EUMs benefit from syntax 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. Execution transparency benefits EUMs and MDE practitioners alike.

There are currently no MTLs explicitly targeting the needs of end-user modelers, nor are there any MTLs simultaneously implementing all three aspects of transparency described here. We address this 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 [46]). VMTL is a model-to-model, unidirectional transformation language supporting endogenous and exogenous transformations, rule application conditions, rule scheduling, and both in-place and out-place transformations. VMTL transformations can be specified for models expressed in any general-purpose or domain-specific modeling language meeting the preconditions defined in Section 4.3.

In this paper we provide a detailed description of VMTL's syntax and operational semantics. We also describe the tool support provided for the language. Since the argument put forward by VMTL is one of superior usability for EUMs, an empirical validation of this claim is required. We have therefore investigated the usability of VMTL – in particular, its learnability – via two user experiments and a pilot think-aloud protocol study.

This paper is structured as follows: Section 2 offers an intuitive understanding of VMTL via a running example from the banking domain, Section 3 presents VMTL's syntax, semantics, and limitations, Section 4 describes tool support, Section 5 presents empirical results regarding VMTL's learnability, Section 6 summarizes related research, and Section 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 [5]. Analysis-level models emphasize quality even further, motivated by compliance to legislation such as the Sarbanes-Oxley Act [2]. In what follows, we illustrate how quality assurance transformations on analysis-level models can be specified using VMTL. For this purpose, consider the Unified Modeling Language (UML [37]) 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.

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 [37], page 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 in VMTL. The core construct of VMTL is the *pattern*, with several pattern types available. A VMTL pattern is always a valid model fragment in the host modeling language, regardless if it is expressed using concrete syntax (as is the case here) or abstract syntax. Transformation 1 consists of a single `Update Pattern` named "Delete Association". Following a successful match in the source model, an `Update Pattern` specifies modifications to this model via textual annotations. In the "Delete Association" pattern, the `delete` annotation specifies that Associations between an Actor and an extending Use Case must be removed from the source model. This 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. The `<<VM Annotation>>` Stereotype is applied in order to distinguish VMTL annotations from regular comments.

VMTL patterns may contain optional visual annotations indicating their type, referred to as *icons*. Such an annotation, implemented as a stereotyped Comment[1], is visible in the top-right corner of the "Delete Association" pattern. While icons help visually identify the type of VMTL patterns, 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 be used to create new model elements, as illustrated by Transformation 2 in Fig. 2 (top left). This transformation ensures that all loan requests require the loan details to be specified. This is accomplished using the `create` annotation 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 redundantly performed several times [2]. 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 `ActivityNode`, an abstract UML meta-class generalizing all Action types, are considered. This is accomplished by an assignment to VMTL's `type` special variable. VMTL also supports *user-defined variables*, such as the `$A` and `$B` variables

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

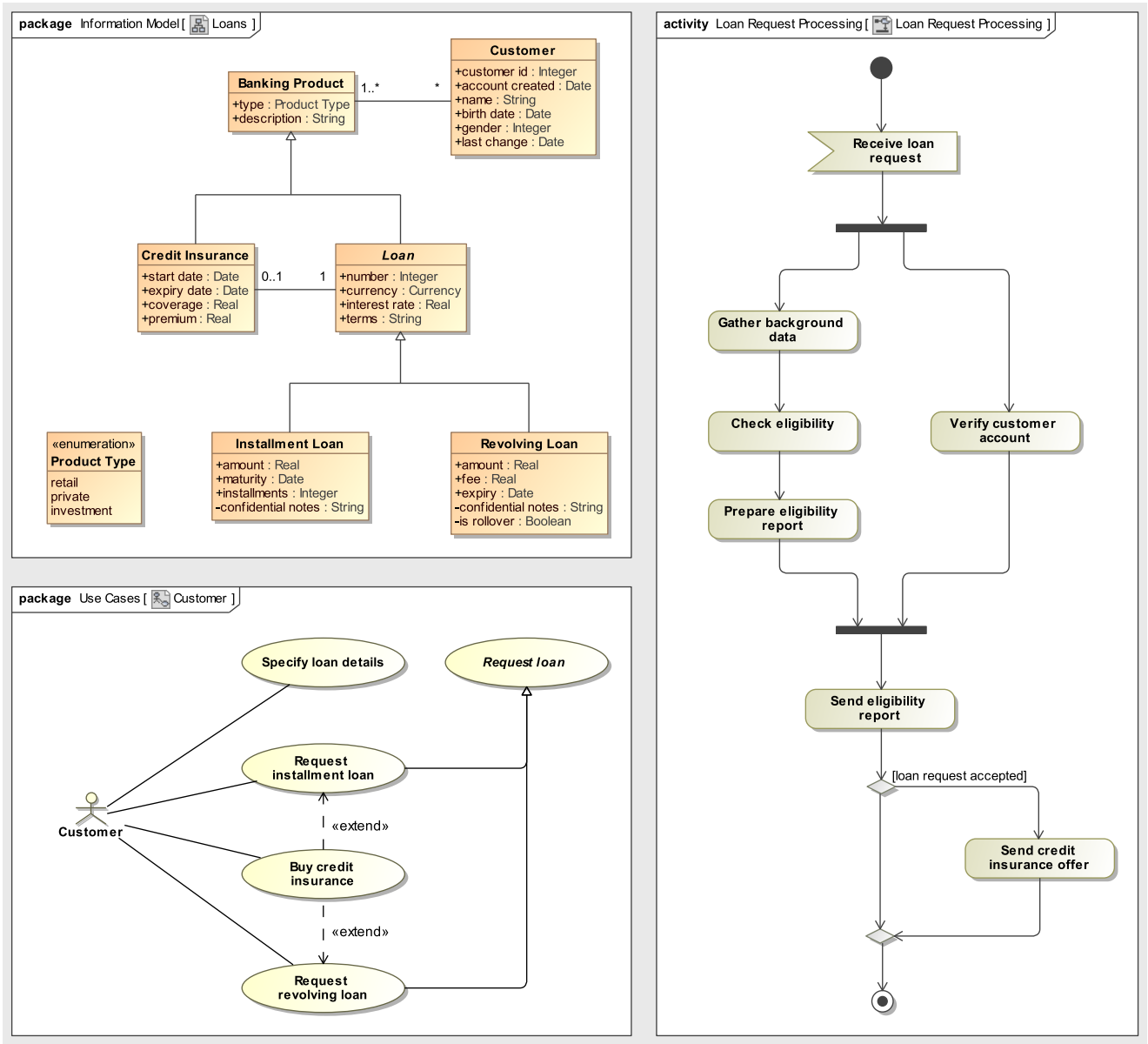

Fig. 1 Analysis-level model representing a financial institution's loan operations. It consists of an information model expressed as a UML Class Diagram (top-left), a use case model expressed as a UML Use Case Diagram (bottom-left), and a process model expressed as a UML Activity Diagram (right).

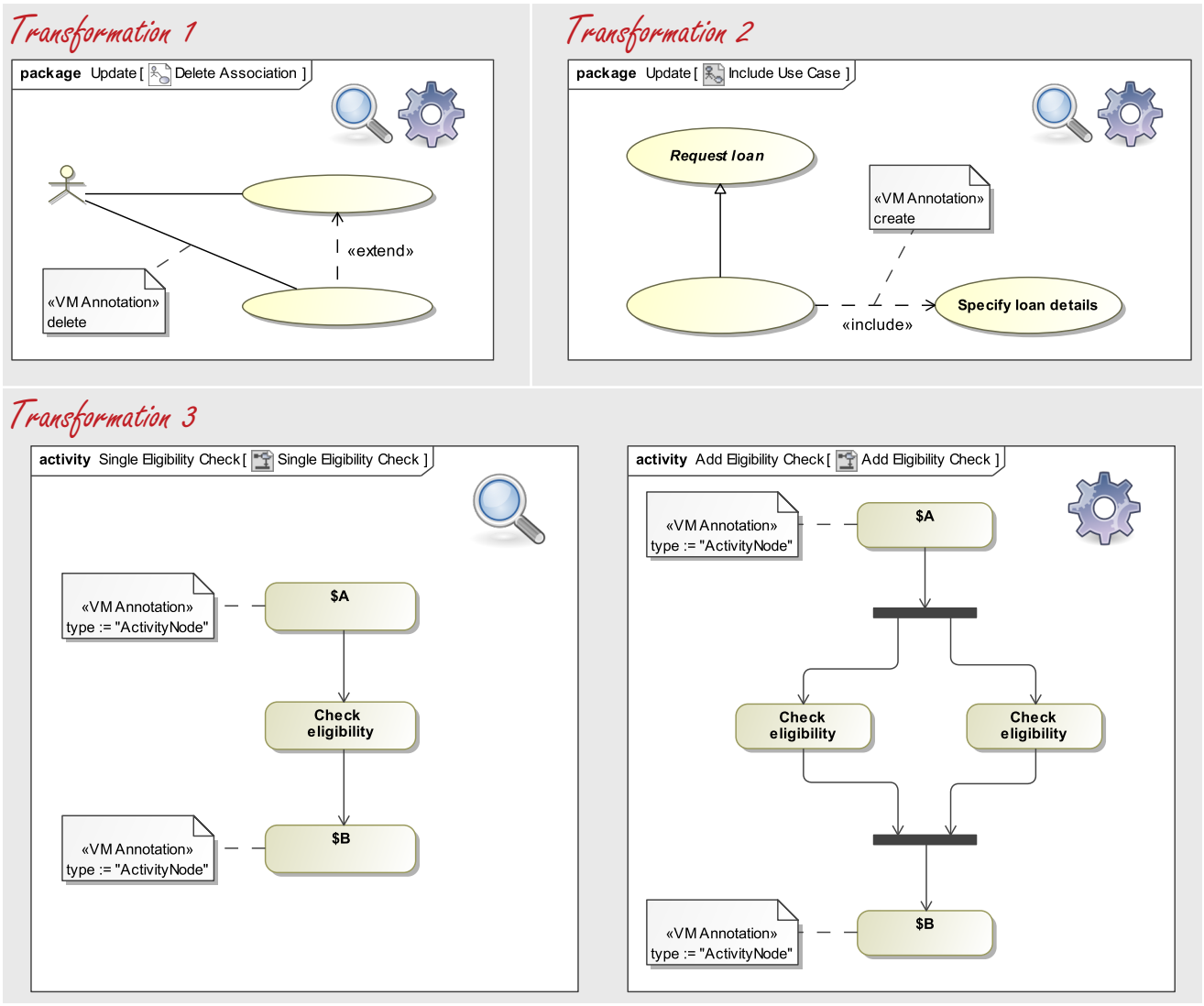

Fig. 2 Single-rule VMTL transformations on the example source model in Fig. 1: Transformation 1 (top-left), Transformation 2 (top-right), Transformation 3 (bottom)

used in the `Find Pattern` in place of the outer Actions' names. When the pattern is matched, these variables are instantiated with the Actions' names ("Gather background data" and "Prepare eligibility report").

Once instantiated, the `$A` and `$B` variables retain their values in the `Produce Pattern`. The differences between the `Produce Pattern` and the `Find Pattern` determine which elements will be created or deleted in the source model. A second Action named "Check eligibility" will be created, together with a Fork/Join Node pair stating that this Action will be executed in parallel with the existing Action of the same name.

Transformation 3 could also be specified using a single `Update Pattern`. All VMTL `Update Patterns` can be expressed as semantically equivalent `Find/Update` pattern pairs, but the opposite is not generally true.

2.3 Multiple-rule transformations

Our final example, the "Pull-Up Attribute" refactoring, addresses a widespread UML Class Diagram design anti-pattern [5]. Its specification states that common Attributes of Classes sharing the same abstract superclass must be deleted, and an Attribute with the same `name`, `type`, and `visibility` (i.e. the same *signature*) must be created in the superclass. In Fig. 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. 3 (top) addresses the basic case with only two subclasses, while Rule 2, shown in Fig. 3 (bottom), handles additional subclasses.

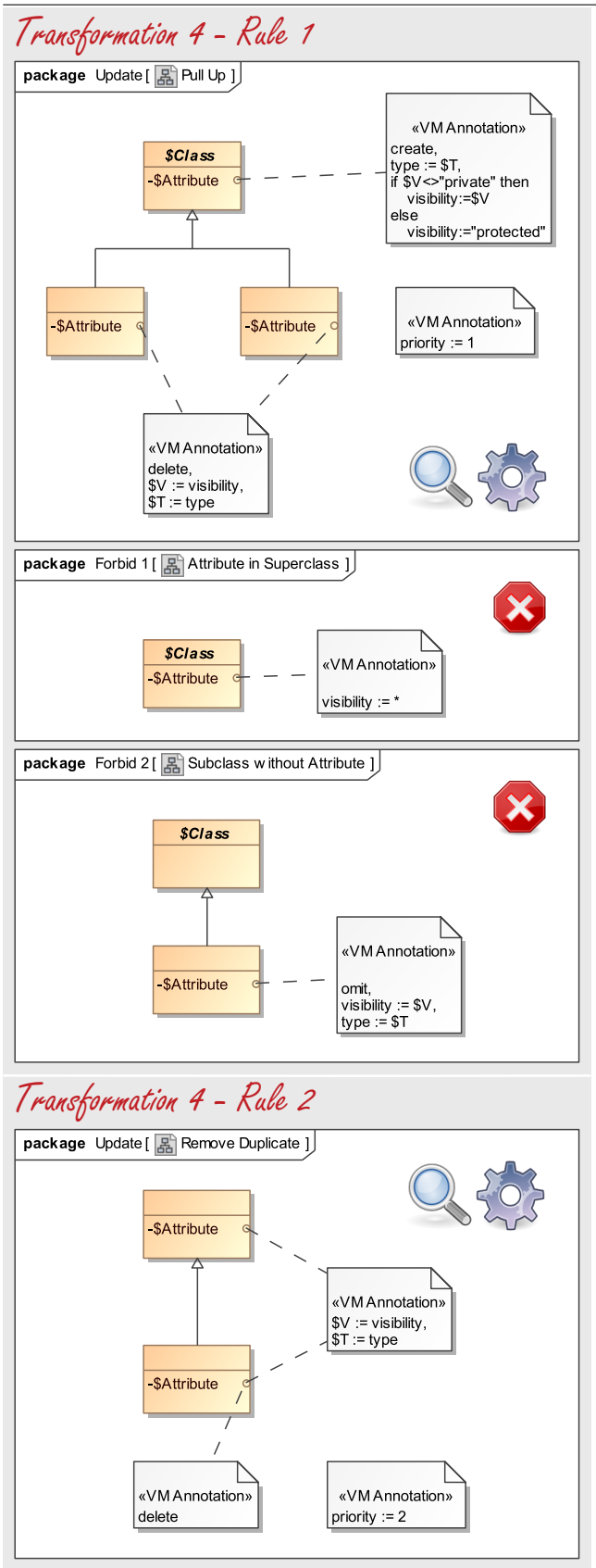

Fig. 3 Multiple-rule VMTL transformation on the example source model in Fig. 1: Transformation 4 – Rule 1 (top), Transformation 2 – Rule 2 (bottom)

Rule 1 contains 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 `$Class` variable, while the `name`, `type`, and `visibility` of the Attribute are stored in the `$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 model element, that element must not appear in a successful match of the pattern.

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.

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 (TMT)*, a collection of three basic 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 Section 6.2), VMTL is the first to follow all three of them.

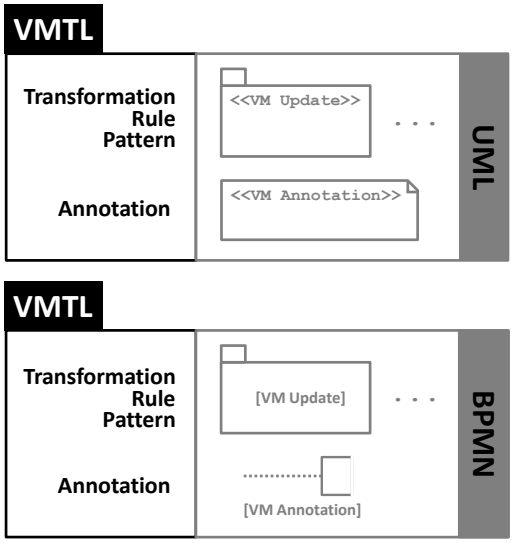

Fig. 4 Realizations of VMTL's structural constructs in UML (top) and BPMN (bottom)

3.1.1 Syntax transparency

The transformation specifications in Fig. 2 and Fig. 3 do not just resemble the concrete syntax of UML diagrams. They are, in fact, valid UML model fragments. Formally, syntax transparency is defined as follows:

Definition 1 An MTL capable of expressing specifications for model transformations operating on source models conforming to a metamodel $\mathcal{MM}_1$ and producing target models conforming to a metamodel $\mathcal{MM}_2$ is said to be *syntax transparent* with respect to $\mathcal{MM}_1$ and $\mathcal{MM}_2$ iff all such specifications conform to both $\mathcal{MM}_1$ and $\mathcal{MM}_2$.

In this definition, $\mathcal{MM}_1$ and $\mathcal{MM}_2$ are referred to as *host metamodels* or *host languages*. The definition also applies to MTLs targeting endogenous transformations, in which case the host metamodels coincide.

VMTL ensures syntax transparency by defining lightweight, conformance-preserving host metamodel extensions. The constructs of VMTL – rules, patterns, and annotations – are mapped to existing host metamodel elements using either metamodel extension mechanism or, if such mechanisms are not available, naming conventions. Fig. 4 illustrates the realizations of VMTL constructs in UML and Business Process Model and Notation (BPMN [34]), respectively. The UML realizations rely on Stereotypes, such as the `<<VM Update>>` Stereotype applied to Packages and the `<<VM Annotation>>` Stereotype applied to Comments. Meanwhile, the BPMN realizations rely on naming conventions, such as the `[VM Update]` prefix for Package names and the `[VM Annotation]` prefix for Text Annotation IDs.

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 defined below.

Definition 2 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 metamodels, 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. 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.

Definition 3 An MTL is *execution transparent* if transformations specified using it can be executed by compilation to one of several *transformation engines* operating at a lower abstraction level.

An implicit pre-condition is that the transformation engine must support a level of expressiveness at least as high as that of the execution transparent MTL.

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 [27]), transformation primitive libraries (e.g. T-Core [52]), or general purpose programming languages enhanced by modeling APIs are all possible.

3.2 Operational semantics

VMTL is a model-to-model transformation language. It supports both *endogenous* and *exogenous* transformations, that is, transformations in which the source and target models conform to the same or different metamodels, respectively [17,31]. VMTL transformations can be executed *in-place* to modify an existing model, as well as *out-place* to produce a new model.

In VMTL, transformation specifications simultaneously conform to their respective host metamodels and to the VMTL metamodel shown in Fig. 5. According to this metamodel, 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(s), typically at the concrete syntax level. Pattern model elements and meta-attributes that do not have a concrete syntax representation are 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 [18]. 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.
- **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.
- **Update Pattern**: A concise representation specifying 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 `Find Patterns` or `Produce Patterns`.

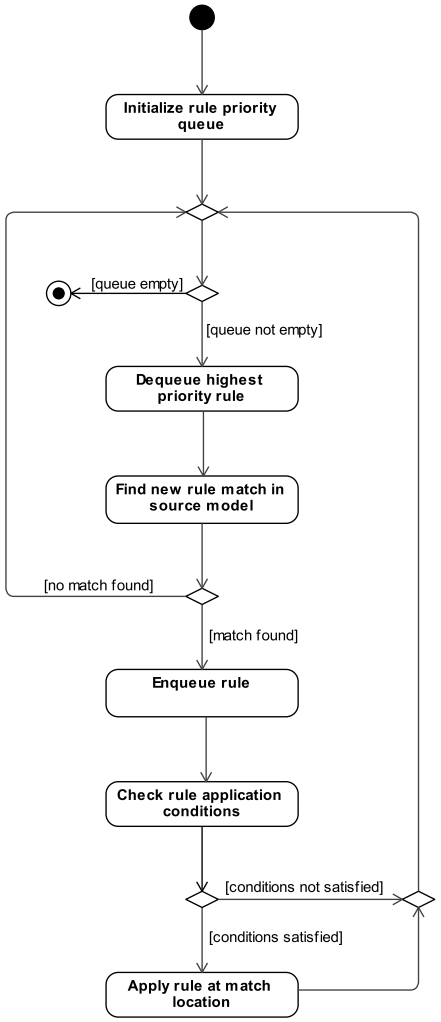

Fig. 6 The VMTL transformation execution process

- **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.

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. 6. The core of this process is a rule *priority queue*.

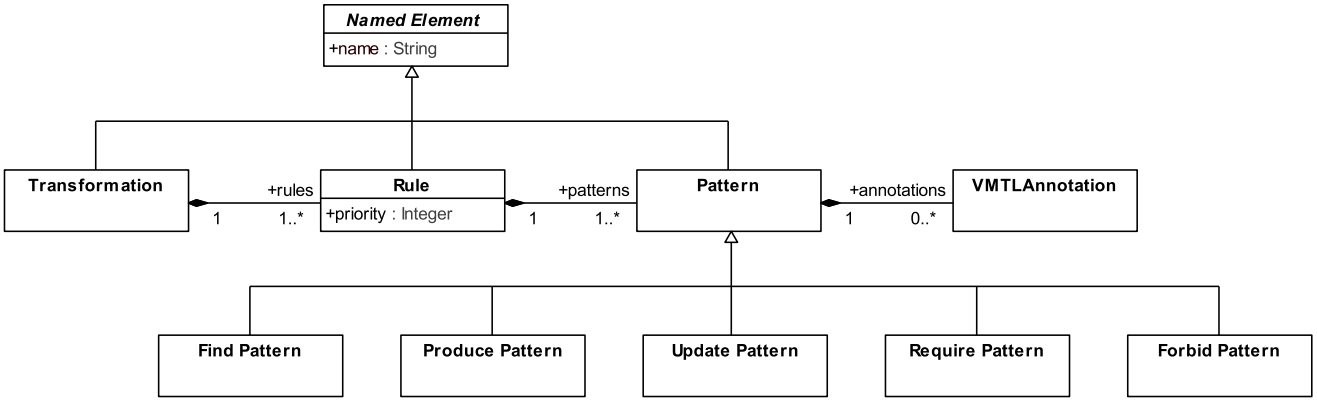

Fig. 5 The VMTL metamodel

While all `Update Patterns` can be expressed as semantically equivalent `Find/Produce Pattern` pairs, the reverse is not true due to two considerations. First, `Update Patterns` cannot specify exogenous transformations, as this would require a single pattern to simultaneously conform to two potentially incompatible metamodels. Second, some endogenous transformations cannot be specified using a single pattern without violating host metamodel constraints. Consider, for instance, a variation of the "Pull-Up Attribute" refactoring shown in Fig. 3. The variation requires existing Attributes in the abstract superclass to be deleted if they have the same name as the Attribute to be pulled-up. The `Update Pattern` in Rule 1 of this transformation would then need to contain two Attributes with the same name, one annotated with the `create` annotation, and one with the `delete` annotation. However, UML does not allow Classes to own attributes with identical names. A possible solution, suggested by Kühne et al. [28], would be to relax the host metamodel constraints. Because this solution breaks the principle of syntax transparency, it is not adopted by VMTL.

Following the principle of syntax transparency, the VMTL metamodel elements in Fig. 5 are mapped to host metamodel 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. Metamodel extension mechanism 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 metamodel 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 metamodel meets the following prerequisites, the first two of which are mandatory:

1. The host metamodel 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 metamodel must include an *annotation element*, such as Comments in UML and Text Annotations in BPMN. All host metamodel elements must support annotations, which act as vehicles for VMTL clauses.
3. The host metamodel 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 model fragments in the host modeling language. Although it supports syntax and environment transparency, this design decision has the

Table 2 The VM* UML profile. This profile is applied to UML Packages representing VMTL transformation specifications.

Stereotype	Extended element	Description	Icon
`<<VM Annotation>>`	Comment	Stereotype applicable to a Comment containing VMTL Annotations	–
`<<VM Transformation>>`	Package	Stereotype applicable to a Package containing a VMTL Transformation	–
`<<VM Rule>>`	Package	Stereotype applicable to a Package containing a VMTL Rule	–
`<<VM Find>>`	Package, Comment	Stereotype applicable to a Package containing a `Find Pattern` or to Comments included in this Package, in which case the Find icon replaces the UML Comment notation.	
`<<VM Produce>>`	Package, Comment	Stereotype applicable to a Package containing a `Produce Pattern` or to Comments included in this Package, in which case the Produce icon replaces the UML Comment notation.	
`<<VM Update>>`	Package, Comment	Stereotype applicable to a Package containing a `Update Pattern` or to Comments included in this Package, in which case the Update icon replaces the UML Comment notation.	
`<<VM Forbid>>`	Package, Comment	Stereotype applicable to a Package containing a `Forbid Pattern` or to Comments included in this Package, in which case the Forbid icon replaces the UML Comment notation.	
`<<VM Require>>`	Package, Comment	Stereotype applicable to a Package containing a `Require Pattern` or to Comments included in this Package, in which case the Require icon replaces the UML Comment notation.	

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 requires relaxing some well-formlessness constraints included in the host metamodel and explicitly referring to elements of the VMTL metamodel.

Under the assumption that the host metamodel must not be modified, VMTL defines a declarative *textual annotation language* used to lift model fragments to the status of model patterns. VMTL annotations support pattern definition, model manipulation, and transformation execution control. As examples, several VMTL annotations are included in Fig. 2 and Fig. 3 as UML Comments carrying the `<<VM Annotation>>` Stereotype.

Dynamically-typed *user-defined variables* may be declared and manipulated in VMTL annotations, and also used as meta-attribute values. The *scope* of such 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 alo accept the *undefined value* ("`*`"). A variable with this value is interpreted as representing any valid value of its respective data type.

As an example of user-defined variable usage, consider Rule 1 of Transformation 4 (Fig. 3). This rule manipulates the `$V` and `$T` variables in its annotations and uses the `$Class` and `$Attribute` variables as meta-attribute values. The values of these variables are set in the rule's `Update Pattern` and read in its two `Forbid Patterns`. However, the similarly-named variables in Rule 2 of the transformation may have different values once this rule is triggered.

For variable manipulation, VMTL supports arithmetic, comparison, and logic operators, as exemplified throughout Fig. 2 and Fig. 3. Logic operators can be invoked through either logic programming-style notations ("`,`", "`;`", "`!`", "`->`") or textual notations ("`and`",

"`or`", "`not`", "`if/then`"). The implication ("`->`") and disjunction ("`;`") operators can be combined to form a conditional "`if/then/else`" construct, as shown in the `Update Pattern` of Rule 1 of Transformation 4 (Fig. 3). 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 done 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 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 leads to 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 implementation. As detailed in this section, our implementation follows all three TMT principles.

4.1 Executing VMTL specifications

The implementation of VMTL is based on the Eclipse Modeling Framework (EMF [45]) and the Henshin [6] transformation engine. We have adopted Henshin due to its graph transformation-based operational semantics, which 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.

The high-level architecture of our implementation, shown in Fig. 7, 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 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. 7 and described in what follows.

In step ❶ 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 ❷ the extracted model fragments are translated into structurally equivalent Henshin graphs intended to play the same role (LHS, RHS, NAC, or PAC) in the generated Henshin transformation. To avoid binding the implementation to a particular modeling language, the fragments are processed in terms of the Ecore meta-metamodel. 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`).

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

Variable	Type	Scope	Description	Examples
`injective`	Boolean	Rule	If set to `true`, 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`
`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 that chain is determined by the value of this special variable.	`steps := 3`, `steps > 3`, `steps := *`
`id`	String	Element	Stores an optional user-defined pattern element identifier in order to facilitate the identification of corresponding elements across patterns.	`id := ‘‘1’’`
`self`	Element	Element	Allows access to the annotated model element.	`self.visibility := ‘‘public’’`
`type`	String	Element	Provides access to the name of the annotated model element's metaclass. Assigning a new value to this special variable modifies the annotated model element's metaclass.	`type := ‘‘Actor’’`

Table 4 Clauses supported by 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 singleton`	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`	Exactly one of the annotated pattern model elements must be matched in the source model.	`Find`, `Forbid`, `Require`
`indirect`	The annotated pattern model element, which must represent a relation, can be matched to a chain of relations of the same type (i.e. its transitive closure) in the source model.	`Find`, `Forbid`, `Require`
`omit`	The annotated pattern model element must not be matched in the source model.	`Find`, `Forbid`, `Require`
`optional`	The annotated pattern model element may or may not be matched in the source model.	`Find`, `Forbid`, `Require`
`unique`	The annotated pattern model element must be unique within its scope (e.g. its package) in the matched 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`

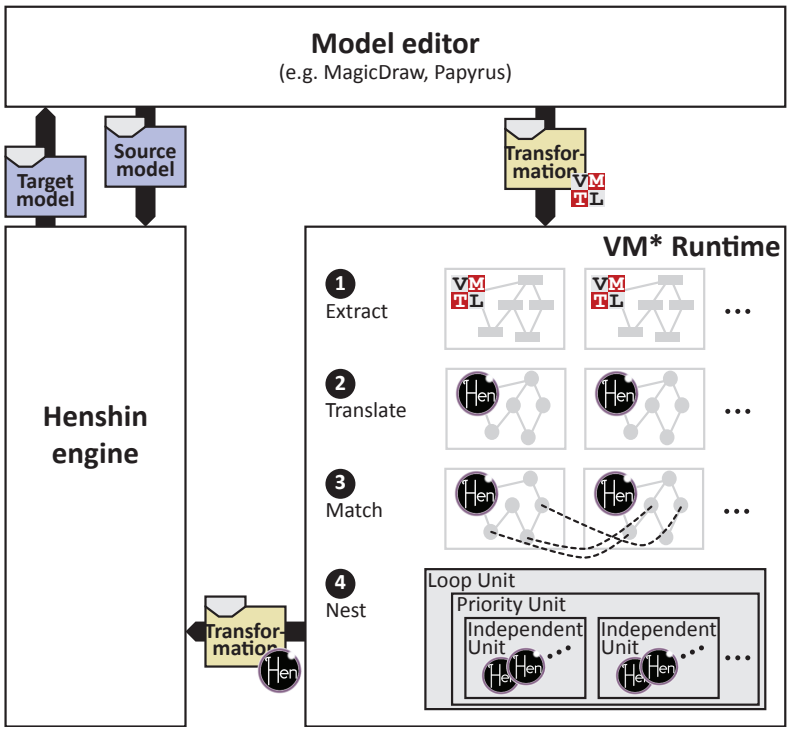

Fig. 7 The architecture of VMTL's implementation. Numbers encircled in black indicate the sequence of steps in the VMTL to Henshin compilation process.

In step ❸ a set of atomic Henshin rules are created by constructing mappings between the nodes of each LHS graph and the corresponding nodes in every other graph belonging to the same rule. As a mapping is a connection between two matching nodes, obtaining the set of mappings between two graphs is equivalent to computing a match between the graphs. The EMF Compare model comparison framework [33] 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 ❹ the generated rules are nested in `Units`, Henshin's control flow formalism. Each Henshin rule is assigned 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 sub-units 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. 7 is amenable to several deployment options. In a monolithic plugin-based deployment, illustrated in Fig. 8 (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[2], as shown in Fig. 8 (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. 8 (bottom), is to forego editor integration entirely and develop a separate Web application as a user interface for VMTL. This

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

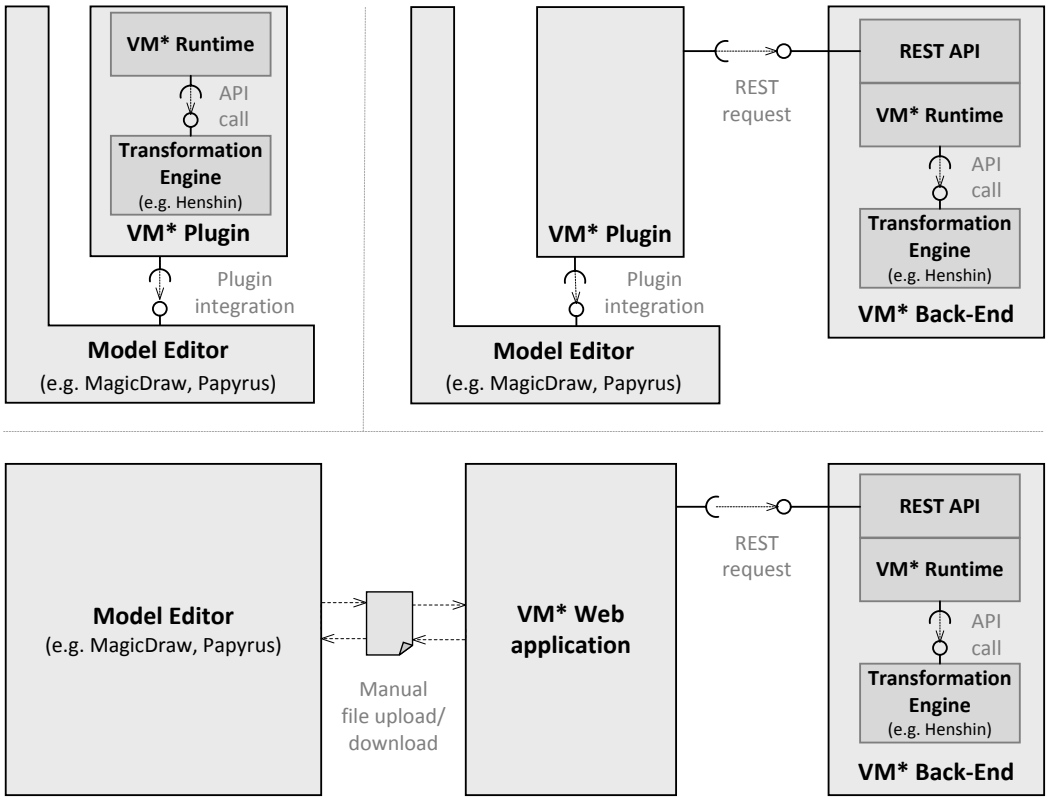

Fig. 8 Deployment options available for VMTL: a self-contained model editor plug-in (top-left), a thin client model editor plug-in (top-right), and a Web application (bottom)

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 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].

4.3 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 metamodel. The expressiveness limitation resulting from this 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 Section 4.2. Finally, displaying target models in the host editor is complicated by the fact that diagram layout is typically not part of the host metamodel. 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 de-

scribed in Section 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 the issue at the language level, by providing the `id` special variable to attach an optional identifier to each pattern element. It is the developer's responsibility to ensure that corresponding elements have the same identifier in all patterns. This element identification provision has the added benefit of allowing the patterns of a rule to conform to different metamodels, thus providing support for exogenous transformations.

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 [14]. 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, with the methodology and outcomes of this evaluation presented in Section 5.1. As a follow-up, we have conducted a think-aloud protocol investigation aiming to discover how users comprehend VMTL specifications and detect shortcomings in the language. The results of this investigation are reported in Section 5.2.

5.1 Learnability experiments

5.1.1 Methods and materials

The purpose of our experiments was to evaluate the *initial learnability* [21] 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.

To this end, we have carried out two questionnaire-based experiments comparing the initial learnability of VMTL with that of a textual MTL (Epsilon) and a visual abstract-syntax MTL (Henshin). Epsilon and Henshin were respectively selected as representing the main transformation paradigms in the MDE landscape: textual hybrid (mostly imperative) languages, and visual graph transformation-based languages [17]. In addition, both Epsilon and Henshin are in widespread use[3].

The designs of our experiments are summarized in Fig. 9. A replication package containing the questionnaires, statistical analysis scripts, and the collected data is available online [1].

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 learning effects, possible participant bias in favor of VMTL as a language developed by their teachers, and lack of result generalizability to the sampled population. The mitigation measures adopted against these threats are described in Section 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 hand-out containing a brief introduction to MT and descriptions of the three MTLs. They were asked to read the hand-out 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.

Each questionnaire's comprehension section contains a total of nine multiple-choice questions, three for every MTL. To answer a question, participants were required

[3] 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`.

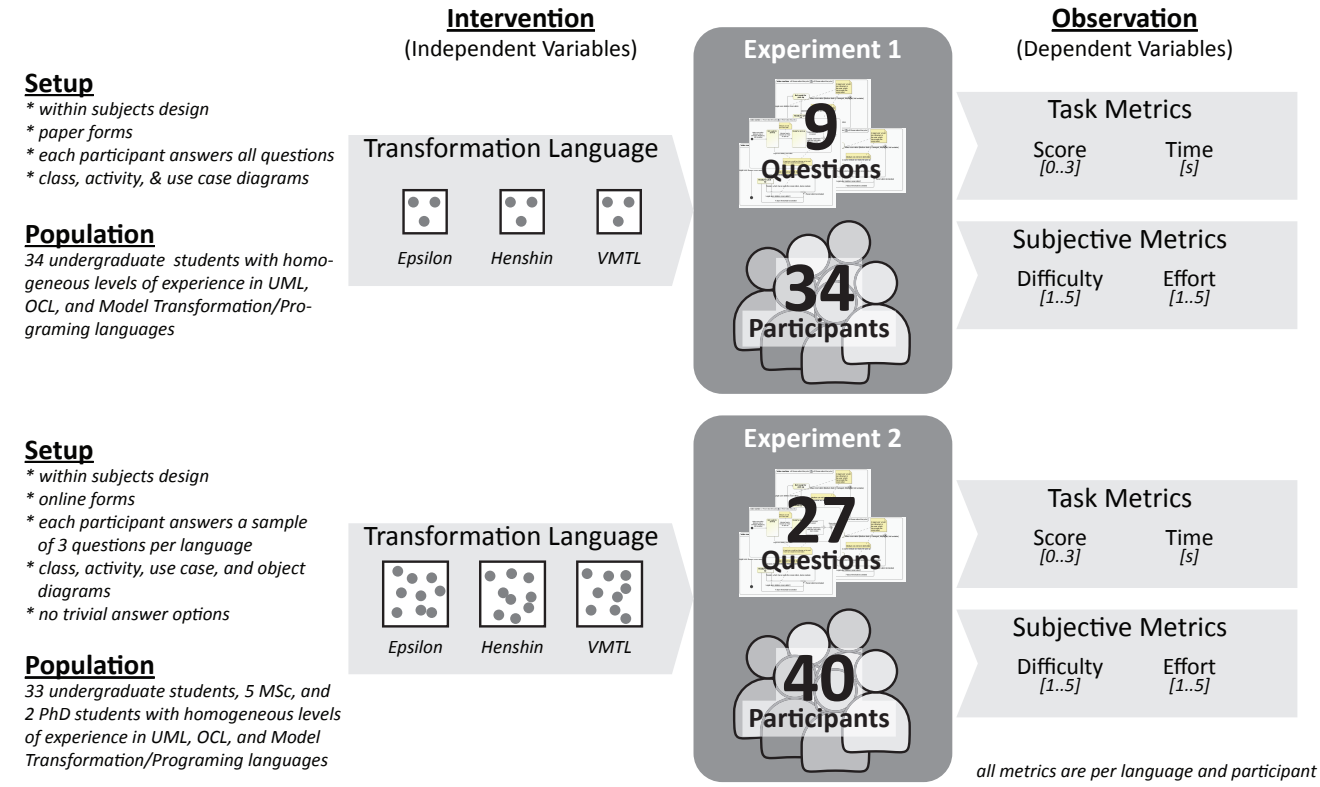

Fig. 9 Overview of the study designs adopted for the learnability experiments

to select the correct natural language description of a given 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 [47] 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).

Table 5 Mean (μ), median (M), and standard deviation (σ) 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 [47].

		Specification size		
		μ	M	σ
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

The example transformations in Fig. 2 and Fig. 3 are representative for the type of transformations included in our experiments, namely in-place model updates as commonly used for software model quality assurance [5]. Transformation 1 in Fig. 2 is in fact used as a question in Experiment 2, which also includes equiva-

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 to 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 to 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.

lent 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.

The adopted hypothesis testing techniques were the Analysis of Variance (ANOVA, [32]) and the Wilcoxon signed rank test. Effect sizes were evaluated using the η^2 statistic in the case of ANOVA[4], and Pearson's r in the case of the Wilcoxon signed rank test[5]. The variance homogeneity and normal distribution of observations required as prerequisites for applying ANOVA were verified, as recommended in the literature [32], using residual plots and Q-Q plots.

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. 10. 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. 10, 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.

In Experiment 1, language is a significant factor for both high- and low-performing participants ($p =$

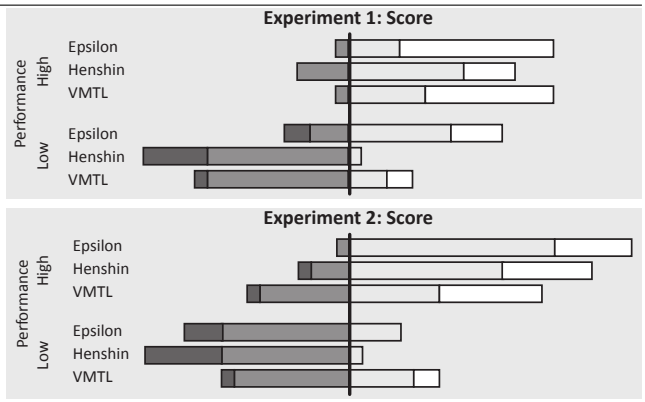

Fig. 10 Stacked bar graphs illustrating comprehension scores for each MTL. Lighter colors correspond to higher scores: white sections show the proportion of participants obtaining the maximum score (3), dark grey sections show the proportion of participants obtaining the minimum score (0).

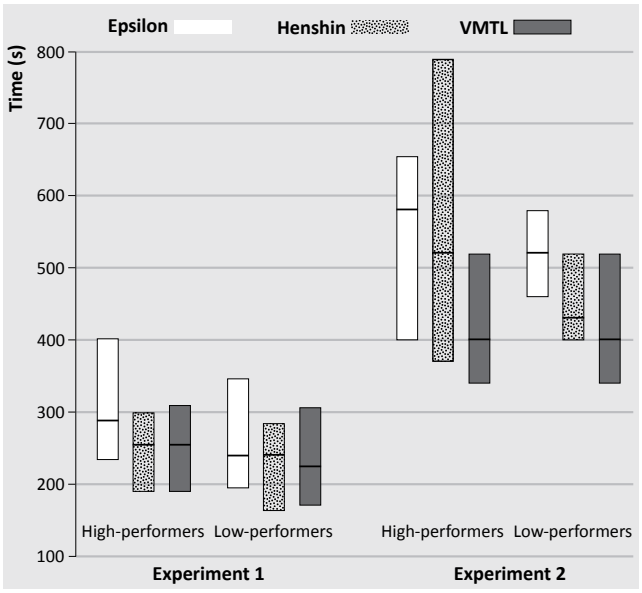

Fig. 11 Box plots illustrating completion times of the comprehension questions for each MTL, grouped by experiment and participant performance

[4] Guidelines suggest that η^2 values greater than 0.06 indicate a moderate effect size, and values greater than 0.14 indicate a large effect size [16].

[5] Values for Pearson's r range in the interval $[-1, 1]$. Values closer to 0 indicate a higher correlation, and therefore a smaller effect size.

Table 7 Mean (μ) and standard deviation (σ) of comprehension scores, comprehension times, and cognitive load ratings for Experiment 1

		Score [0..3]		**Time** (s)		**Difficulty** [1..5]		**Effort** [1..5]	
		μ	σ	μ	σ	μ	σ	μ	σ
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 (μ) and standard deviation (σ) of comprehension scores, comprehension times, and cognitive load ratings for Experiment 2

		Score [0..3]		**Time** (s)		**Difficulty** [1..5]		**Effort** [1..5]	
		μ	σ	μ	σ	μ	σ	μ	σ
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

0.0117 and $p = 7.13e - 4$, respectively). However, effect size is larger for low-performers ($\eta^2 = 0.229$) than for high-performers ($\eta^2 = 0.1672$). In the case of high-performers, both Epsilon ($p = 0.0146$, $r = 0.1457$) and VMTL ($p = 0.033$, $r = 0.1416$) are associated to 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 ($p = 0.0047$, $r = -0.0825$) and VMTL ($p = 0.037$, $r = 0.0831$) are associated to significantly higher scores compared to Henshin, while the difference between scores obtained under Epsilon and VMTL is not statistically significant ($p = 0.163$). On the other hand, in Experiment 2, language only has a statistically significant effect on comprehension scores for low-performing participants ($p = 0.0214$, $\eta^2 = 0.1798$). For this participant group, VMTL is associated to significantly higher scores than Henshin ($p = 0.0186$, $r = -0.2189$), while other score differences are not statistically significant.

Completion times for the comprehension task are shown in the second data columns of Table 7 and Table 8, respectively. They are illustrated as box plots in Fig. 11. A first observation is that completion times are longer for Experiment 2, which features slightly more complex transformations. In terms of the effect of language, Experiment 1 participants seem to have 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 ($p = 0.0558$, $\eta^2 = 0.1185$), 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 of high-performers in Experiment 2 are highly dependent on language ($p = 0.0072$, $\eta^2 = 0.2079$). Here, VMTL is associated with shorter completion times than both Epsilon ($p = 0.0694$) and Henshin ($p = 0.025$, $r = 0.2081$), while Epsilon is possibly associated with shorter completion times than Henshin ($p = 0.0587$).

Difficulty and effort ratings are summarized in the rightmost data columns of Table 7 and Table 8, and illustrated in Fig. 12 as stacked bar graphs. The bars in Fig. 12 are based on a 5-point scale of possible rating values. Lighter colors correspond to higher difficulty and effort ratings.

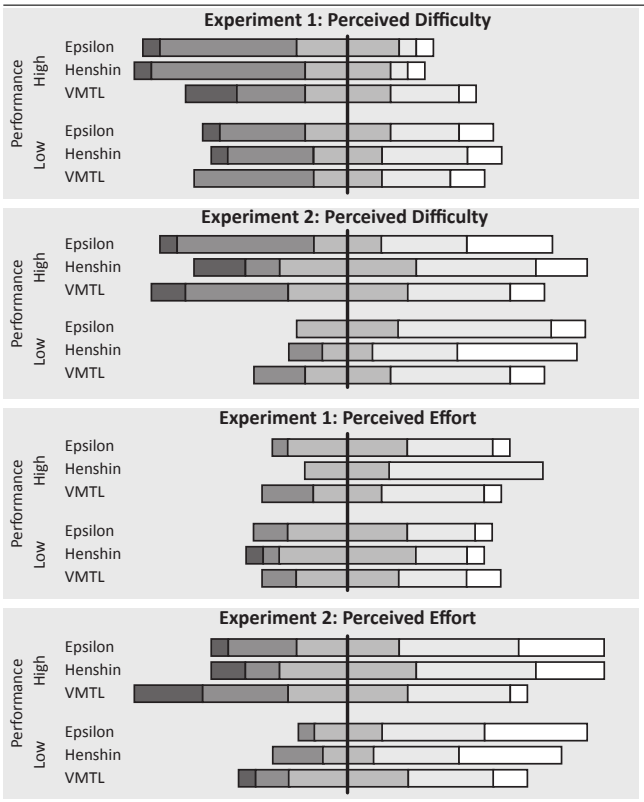

Fig. 12 Stacked bar graphs illustrating cognitive load ratings for each MTL. Lighter colors correspond to higher ratings: white sections show the proportion of participants assigning the maximum rating (5), dark grey sections show the proportion of participants assigning the minimum rating (1).

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 ($p = 0.2345$ and $p = 0.3297$ 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 are the potentially significantly higher difficulty ratings assigned by high-performing participants to Henshin compared to VMTL ($p = 0.0512$, $r = -0.1506$).

For Experiment 1, effort ratings 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 ($p = 0.7525$ and $p = 0.4436$ when compared to Epsilon and VMTL, respectively). The only statistically significant impact of language on any of the cognitive load measurements is registered for the effort ratings of Experiment 2. High-performing participants rate VMTL as requiring significantly less effort than Henshin ($p = 0.0092$, $r = 0.3513$), and potentially significantly less effort than Epsilon ($p = 0.1049$, $r = 0.1792$). A similar trend, but lacking statistical significance, is observed for low-performers.

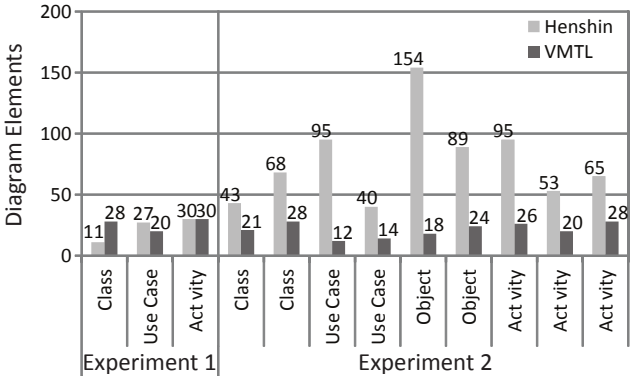

Fig. 13 Number of diagram elements (shapes, line segments, labels) included in the Henshin and VMTL specifications presented to participants

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 associated to 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 increase usability. This is achieved by employing expressive visual notations and hiding unessential metamodel details. A second explanation for VMTL's better performance has to do with specification size. As shown in Figure 13, 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 [47].

The high comprehension scores associated to 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 to Epsilon is offered by *cognitive fit theory* [59], which has primarily been applied in the field of information visualization [55,58]. 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.

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. [62].

Construct validity. An experiment manifests construct validity if it measures the actual phenomena under investigation – in our case, model transformation language learnability. Although we use subjective cognitive load measures, it has been shown that a high correlation exists between subjective and objective cognitive load measures (cf. [19]). In addition, since VMTL is developed by the authors of this study (to the best of our knowledge, participants were not aware of this), 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. 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 non-parametric 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. [32].

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 participants 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.

5.2 Think-aloud protocol

5.2.1 Methods and materials

While our experiments offer an objective measure of VMTL's relative learnability when compared to Epsilon and Henshin, they do not provide any insights into how users approached the task of comprehending a VMTL

specification. To this end, we have performed a pilot concurrent think-aloud protocol investigation [30].

Four Ph.D. students with diverse backgrounds (a nutritionist, a theoretical computer scientist, and two software engineers) took part in this study. They were each presented with a subset of the VMTL transformations included in Experiment 2, namely Questions 2, 3, and 7. The chosen transformations respectively operate on Class Diagrams (Question 2), Use Case Diagrams (Question 3), and Activity Diagrams (Question 7). After sitting through an introduction to VMTL, participants were asked to read the three VMTL specifications and explain their intended meaning. They were allowed to consult a written specification of VMTL at any time.

5.2.2 Results

This study has yielded several interesting findings. First, it has confirmed that allowing VMTL transformation developers to use either `Update Patterns` or equivalent `Find/Replace Pattern` pairs is a positive 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 offered by `Update Patterns`. In contrast, 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 an 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. The first could not precisely identify if an annotation refers to an Attribute or to its containing Class, when the annotation in fact referred to the Class. The second was surprised that an annotation can be anchored to an Association, and could not determine which end of the Association it refers to – in fact, the annotation referred to the Association itself. Finally, one participant repeatedly referred to the annotations as *"log messages"*. These preliminary observations suggest that textual annotations may not be as intuitive as hoped.

The `create singleton` clause has also caused participants some difficulty. They repeatedly consulted its written description, and one participant described it as *"taking an existing model element and turning it into a singleton"*. The clause's intended semantics 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 consider replacing this clause with a more explicit formulation, such as `create if not exists`.

While the limited scale of this pilot study prevents us from drawing any final conclusions, it has provided a clear indication of the aspects of VMTL that can be improved. It has also suggested possible contributing factors to VMTL's comprehension scores in Experiments 1 and 2. We intend to use this information in subsequent experimental assessments, as part of our effort to iteratively improve VMTL's usability.

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 [25]) and Join Point Designation Diagrams (JPDD [44]) allows users to express queries as concrete syntax patterns. The same technique can be applied to specify model constraints [49]. The business process modeling community particularly emphasizes end-user model querying through languages such as BP-QL [12] and BPMN-Q [8].

VMTL itself has been developed on the foundation of VMQL [46], a usability focused query-by-example language. Like VMTL, VMQL has also been experimentally evaluated. It has been shown to offer superior usability compared to OCL, the de-facto standard model querying language, when querying UML models [46] and BPMN models [48].

6.2 Transparent Model Transformation

The insight that model transformations can be viewed as models [13] has been the first step towards syntax transparency. Early MTLs such as VMT [42] and MOLA [24] 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 [53], take a similar route. However, these approaches stop short of achieving syntax transparency, as they augment their host languages' concrete syntax. The augmentation purpose is typically to provide expressive rule execution control mechanisms such as flowcharts [24,53].

A number of existing MTLs do, however, adhere to a subset of the transparency principles defined in Section 3.1 – albeit without explicitly claiming this. These MTLs are listed in Table 9.

Table 9 Approaches supporting explicit MT specifications and exhibiting syntax transparency (ST), environment transparency (EnT), or execution transparency (ExT)

Approach	ST	EnT	ExT
AToMPM [53]	✘	✘	✔
MATA [60]	✔	✘	✔
MeTAGeM [15]	✘	✘	✔
MoTif [51]	✘	✘	✔
MoTMoT [56]	✘	✔	✔
PICS [9]	✔	✘	✘
QVT-R [36]	✘	✘	✔
Schmidt [39]	✔	✘	✘
transML [22]	✘	✘	✔
VMTL	✔	✔	✔

Syntax transparency is exhibited by MATA [60], PICS [9] and the MTL proposed by Schmidt [39], 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 both environment and 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 [54] 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.

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 [56], 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 [52], along with their usage for implementing existing MTLs such as MoTif [51], 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 [22] and MeTAGeM [15] tools.

A separate model transformation paradigm aimed at leveraging concrete syntax is Model Transformation By-Example (MTBE [57]). 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 [7, 10, 26, 38, 61], users explicitly state the correspondences between example source and target model elements. In *demonstration-based* approaches [29, 50], 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 9 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.

Batory et al. [11] adopt learnability as the main design goal of their MTL proposal. Following a failed attempt to teach MT using EMF, the authors state that the EMF toolchain promotes *"incantations to solve problems"*, and consequently propose their own MTL in the form of a library for the Prolog logic programming language. However, no evidence is provided to support the claimed learnability improvements.

All existing empirical evidence regarding MTL usability is of a qualitative nature. Silva et al. [43] 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 selection of evaluated MTLs is discussed, and language learnability is not addressed. Grønmo et al. [20] investigate the usability of three MTLs by comparing the conciseness and required development effort for 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.

The single existing user study evaluating the usability of a MT approach addresses the CONVErT MTBE framework [7]. The presented evaluation is purely qualitative, and features a total of 15 participants which are asked to complete a transformation task using CONVErT and provide a subjective account of their experience via a questionnaire. Most participants rate the framework as easy to learn and use, but data describing their objective performance is not provided. No other MTLs are considered, and the authors make no claim for the generalizability of their results. This leaves the evaluation of VMTL presented in Section 5.1 as the only experimental investigation of MTL usability.

7 Conclusions

7.1 Summary

End-user modelers are domain experts that create and update models as part of their work. They are expected to be closely familiar with some particular modeling languages, but have no incentive to master metamodeling 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 metamodel, 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 have 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 address this problem by creating an MTL for end-user modelers.

We have adopted a systematic approach to developing such an MTL. First, we have investigated its requirements 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 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 metamodel extension that does not break compatibility with existing model editors. Transformation specifications created this way can then be executed by any sufficiently expressive transformation 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 usability, an argument requiring empirical confirmation. We have therefore conducted complementary empirical investigations into VMTL's usability, 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 to 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 performed a pilot concurrent think-aloud protocol study. 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 provided the first characterization of end-user modelers, and compared their modeling-related skills to those of MDE practitioners. Based on the principles of Transparent Model Transformation, which we have defined in a previous publication, we have proposed VMTL as the first model transformation language explicitly addressing the needs and capabilities EUMs. 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 pilot think-aloud protocol study aimed at uncovering potential shortcomings in the language.

7.3 Future work

We intend to continuously improve VMTL as we obtain additional empirical evidence regarding its usability. Such improvements are best informed by qualitative evidence, motivating us to emphasize interviews and think-aloud protocol studies in the future. The results of our pilot think-aloud protocol study presented in this paper 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, a variation of the presented experiments including participants with no computer programming background may 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 also be mitigated by providing visual answer options for the comprehension questions. Finally, studying additional MTLs would allow us to either more confidently generalize or re-consider our conclusions.

The extension of VMTL's tool support is an equally important future work direction. VMTL specifications can currently only be executed using the Henshin transformation engine. Extending the VM* Runtime to accommodate additional execution engines will lend credibility to VMTL's syntax transparency claims. We will also leverage VMTL's service-based deployment to reach end-user modelers by creating lightweight model transformation plugins for widely used traditional model editors such as Microsoft Visio and MagicDraw.

References

1. VMTL Experimental Replication Package. `https://vmstar.compute.dtu.dk/doku.php?id=vmtl:evaluation`
2. Sarbanes-Oxley Act. 107th Congress Public Law 204, U.S. Government Printing Office (2002)
3. Acretoaie, V., Störrle, H.: Hypersonic: Model Analysis and Checking in the Cloud. In: Proc. 2nd Workshop on Scalability in Model Driven Engineering, *CEUR Workshop Proceedings*, vol. 1206, pp. 6–13 (2014)
4. 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 International Publishing (2015)
5. Arendt, T.: Quality Assurance of Software Models. Ph.D. thesis, Philipps-Universität Marburg (2014)
6. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Model Driven Engineering Languages and Systems, *LNCS*, vol. 6394, pp. 121–135. Springer Berlin Heidelberg (2010)
7. 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)
8. Awad, A., Sakr, S.: On efficient processing of BPMN-Q queries. Comp. Ind. **63**(9), 867–881 (2012)
9. Baar, T., Whittle, J.: On the Usage of Concrete Syntax in Model Transformation Rules. In: PSI 2006, *LNCS*, vol. 4378, pp. 84–97. Springer Berlin Heidelberg (2007)
10. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Softw. Syst. Model. **8**(3), 347–364 (2009)
11. Batory, D., Latimer, E., Azanza, M.: Teaching Model Driven Engineering from a Relational Database Perspective. In: Model-Driven Engineering Languages and Systems, *LNCS*, vol. 8107, pp. 121–137. Springer Berlin Heidelberg (2013)
12. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes with BP-QL. Inf. Syst. **33**(6), 477–507 (2008)
13. 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 Heidelberg (2006)
14. 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)
15. Bollati, V.A., Vara, J.M., Jiménez, Á., Marcos, E.: Applying MDE to the (semi-)automatic development of model transformations. Inform. Software Tech. **55**(4), 699–718 (2013)
16. Cohen, J.: Statistical Power Analysis for the Behavioral Sciences, second edn. Routledge (1988)
17. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Syst. J. **45**(3), 621–645 (2006)
18. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer Berlin Heidelberg (2006)
19. Gopher, D., Braune, R.: On the psychophysics of workload: Why bother with subjective measures? Human Factors **26**(5), 519–532 (1984)
20. Grønmo, R., Møller-Pedersen, B., Olsen, G.K.: Comparison of Three Model Transformation Languages. In: Model Driven Architecture – Foundations and Applications, *LNCS*, vol. 5562, pp. 2–17. Springer Berlin Heidelberg (2009)
21. Grossman, T., Fitzmaurice, G., Attar, R.: A Survey of Software Learnability: Metrics, Methodologies and Guidelines. In: Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'09), pp. 649–658. ACM, New York, NY, USA (2009)
22. 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)
23. Jones, B., Kenward, M.G.: Design and Analysis of Cross-Over Trials, third edn. Chapman and Hall/CRC (2014)
24. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. In: Model Driven Architecture, *LNCS*, vol. 3599, pp. 62–76. Springer Berlin Heidelberg (2005)
25. Kent, S.: Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In: Proc. 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97), pp. 327–341. ACM (1997)
26. Kessentini, M., Sahraoui, H., Boukadoum, M., Omar, O.B.: Search-based Model Transformation by Example. Softw. Syst. Model. **11**(2), 209–226 (2012)

27. 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 Heidelberg (2006)
28. 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 Heidelberg (2010)
29. 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 Heidelberg (2010)
30. Lewis, C.: Using the "Thinking Aloud" Method In Cognitive Interface Design. Tech. Rep. RC-9265, IBM (1982)
31. Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science **152**, 125–142 (2006)
32. Montgomery, D.C.: Design and Analysis of Experiments, eighth edn. John Wiley & Sons (2012)
33. Obeo: EMF Compare. `https://www.eclipse.org/emf/compare/`
34. Object Management Group: Business Process Model and Notation (BPMN) 2.0.2. OMG Document formal/2013-12-09 (2013)
35. Object Management Group: Object Constraint Language 2.4. OMG Document formal/2014-02-03 (2014)
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: Unified Modeling Language (UML) 2.5. OMG Document formal/2015-03-01 (2015)
38. 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 Heidelberg (2012)
39. 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 Heidelberg (2007)
40. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Softw. **20**(5), 19–25 (2003)
41. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Softw. **20**(5), 42–45 (2003)
42. 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)
43. Silva, G.C., Rose, L.M., Calinescu, R.: A Qualitative Study of Model Transformation Development Approaches: Supporting Novice Developers 18-27. In: Proc. Intl. Workshop on Model-Driven Development Processes and Practices (MD2P2'14), *CEUR Workshop Proceedings*, vol. 1249, pp. 18–27 (2014)
44. 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)
45. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, second edn. Addison-Wesley Professional (2008)
46. Störrle, H.: VMQL: A Visual Language for Ad-Hoc Model Querying. J. Visual Lang. Comput. **22**(1) (2011)
47. 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 International Publishing (2014)
48. Störrle, H., Acretoaie, V.: Querying Business Process Models with VMQL. In: Proc. ACM SIGCHI Annual International Workshop on Behaviour Modelling - Foundations and Applications (BMFA'13), pp. 4:1–4:10. ACM, New York, NY, USA (2013)
49. 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 Heidelberg (2009)
50. Sun, Y., White, J., Gray, J.: Model Transformation by Demonstration. In: Model Driven Engineering Languages and Systems, *LNCS*, vol. 5795, pp. 712–726. Springer Berlin Heidelberg (2009)
51. Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. Softw. Syst. Model. **12**(2), 387–414 (2011)
52. Syriani, E., Vangheluwe, H., LaShomb, B.: T-Core: a framework for custom-built model transformation engines. Softw. Syst. Model. **13**(3), 1–29 (2013)
53. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Huseyin, E.: AToMPM: A Web-based Modeling Environment. In: Joint Proc. of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition, *CEUR Workshop Proceedings*, vol. 1115, pp. 21–25 (2013)
54. 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 Heidelberg (2004)
55. 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. Visualization and Computer Graphics **16**(5), 841–853 (2010)
56. 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 Heidelberg (2009)
57. Varró, D.: Model Transformation By Example. In: Model Driven Engineering Languages and Systems, *LNCS*, vol. 4199, pp. 410–424. Springer Berlin Heidelberg (2006)
58. Vessey, I.: Cognitive Fit: A Theory-Based Analysis of the Graphs Versus Tables Literature. Decision Sciences **22**(2), 219–240 (1991)
59. 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)
60. 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 Heidelberg (2009)
61. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards Model Transformation Generation By-Example. In: Proc. 40th Annual Hawaii International Conference on System Sciences, HICSS'07, pp. 285b–. IEEE Computer Society, Washington, DC, USA (2007)
62. Wohlin, C., Runeson, P., Höst, M., C. Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer Berlin Heidelberg (2012)