

VM*: A Family of Visual Model Manipulation Languages



Harald Störrle and Vlad Acrețoiaie

Abstract CONTEXT: Practical facilities for querying, constraining, and transforming models (“model management”) can significantly improve the utility of models, and modeling. Many approaches to model management, however, are very restricted, thus diminishing their utility: they support only few use cases, model types or languages, or burden users to learn complex concepts, notations, and tools. GOAL: We envision model management as a commodity, available with little effort to every modeler, and applicable to a wide range of use cases, modeling environments, and notations. We aim to achieve this by reusing the notation for modeling as a notation for expressing queries, constraints, and transformations.

METHOD: We present the VM* family of languages for model management. In support of our claim that VM* lives up to our vision we provide as evidence a string of conceptual explorations, prototype implementations, and empirical evaluations carried out over the previous twelve years.

RESULTS: VM* is viable for many modeling languages, use cases, and tools. Experimental comparison of VM* with several other model querying languages has demonstrated that VM* is an improvement in terms of understandability. On the downside, VM* has limits regarding its expressiveness and computational complexity.

CONCLUSIONS: We conclude that VM* largely lives up to its claim, although the final proof would require a commercial implementation, and a large-scale industrial application study, both of which are beyond our reach at this point.

H. Störrle (✉)
QAware GmbH , München, Germany
hstorle@acm.org
e-mail: harald.stoerrle@ifi.lmu.de

V. Acrețoiaie
FREQUENTIS Romania SRL , Cluj-Napoca, Romania

1 Introduction

The *Visual Model Manipulation Language* VM* is a lineage of languages that allows to express queries (VMQL), constraints (VMCL), and transformations (VMTL); refer to Table 1. Figure 1 illustrates the relationship between these three languages. VM* aspires to be truly useful, addressing the problems faced by the working modeler. Thus, our design goals are usability and learnability, versatility and practical use cases, coverage of all relevant visual modeling notations and compatibility with existing modeling environments. As a consequence, there is a limit to the expressiveness and the scope of application scenarios VM* addresses. For instance, VM* is not suited to process ultra large models, or express very complex model transformations. In our experience, those situations are rare.

Starting out as a query language, VM* has evolved into a full blown model manipulation language that can also handle constraints and transformations. Obvi-

Table 1 Main publications on VM* and its precursors, most notably VMQL and VMTL. In column “Intent”, Q, C, and T refer to queries, constraints, and transformations, respectively, whereas bullets indicate the types of instructions addressed in the corresponding publications. In column “Type”, W, C, J, and TR stand for **W**orkshop, **C**onference, **J**ournal paper, and **T**echnical Report, respectively. BSc, MSc, PhD refer to theses of the respective types. References [7, 32, 34] are re-publications, posters, and excerpts

Year	Ref.	Type	Intent			Title (abbreviated)
			Q	C	T	
2005	[24]	TR	•	•		MoMaT: A lightweight platform for MDD
2007	[25]	W	•			A PROLOG approach to representing & querying models
2009	[39]	BSc	•			MQ: A visual query-interface for models
	[26]	W	•			A logical model query-interface
	[27]	C	•			VMQL: A generic visual model query language
2011	[28]	C		•		Expressing model constraints visually with VMQL
2012	[29]	J	•			VMQL: A visual language for ad-hoc model querying
	[3]	MSc	•			An implementation of VMQL
	[5]	W	•			MQ-2: A tool for prolog-based model querying
2013	[30]	W	•			Improving the usability of OCL as an ad-hoc MQ language
	[31]	W	•			MOCQL: A declarative language for ad-hoc model querying
	[35]	W	•			Querying business process models with VMQL
2014	[6]	W	•			Efficient model querying with VMQL
	[8]	W	•	•		Hypersonic: Model analysis and checking in the cloud
2015	[33]	J	•	•		Cost-effective evolution of prototypes: The MACH case study
2016	[10]	J	•	•	•	VMTL: A language for end-user model transformation
	[9]	C	•	•	•	Model transformation for end-user modelers with VMTL
	[4]	PhD	•	•	•	Model manipulation for end-user modelers
	[2]	WIKI	•	•	•	The VM* Wiki

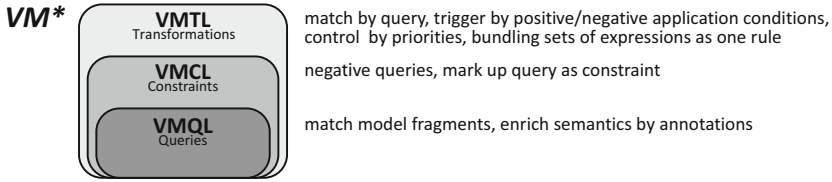


Fig. 1 VM* is a family of model manipulation languages that allows to express queries (VMQL), constraints (VMCL), and transformations (VMTL)

ously, there is a natural succession arising out of the symmetry between queries and constraints. Then, transformations are just pairs of an application condition and a consequence, both of which can be expressed like queries. In that sense, there is not just a natural symmetry but also a great practical opportunity by leveraging queries to constraints and transformations. More importantly, however, there is a practical need for constraints and transformations once queries are established, like progressing from text search to search-and-replace. Model querying is useful, but it is incomplete without checks and transformations. From a high-level point of view, we can characterize queries, constraints, and transformations as follows.

Queries. Assume that a model \mathcal{M} is a set of model elements ME . A Boolean property $\alpha : \mathcal{P}(ME) \rightarrow bool$ characterizes those parts of \mathcal{M} that satisfy α . So, applying α to all fragments of \mathcal{M} amounts to querying \mathcal{M} , and the set of all fragments R_α that satisfy α is the result of the query.

Constraints. Conversely, the dual of R_α are those model fragments of \mathcal{M} that do not satisfy α . If we formulate α just so that it yields all *admissible* fragments of \mathcal{M} , and look at the complement of the result, we have a constraint. So, we need a way of choosing whether we consider R_α or $\overline{R_\alpha}$ as the result. Also, we need to express the complement, or, more generally, a form of negation so that constraints may be expressed in a concise way.

Transformations. Similarly, model transformations are sets of rules, each of which consists of an application condition (“left-hand side”) and a consequence (“right-hand side”). The application condition is again a query or a constraint, while the right-hand side must express matching and changing, i.e., it is a query with side effects.

The general idea of VM* is to reuse the syntax of the modeling language at hand (the *host language*) as the syntax of the query, constraint, or transformation language, adding only a small set of textual annotations. Then, a query is matched against model fragments of the host language based on structural similarity. In the process, annotations are evaluated. Any matching fragments are presented as results. Conversely, when evaluating a constraint, fragments not matching the constraint are presented as violations. For transformations, two parts have to be provided per transformation rule, expressing the left-hand side and right-hand side of the rule. Left-hand side expressions are effectively queries and/or constraints. If it is satisfied,

the right-hand side is executed, either replacing or modifying the matched fragment. The unique approach of VM* provides three main benefits.

Syntax transparency. Queries, constraints, and transformations are expressed using the host language, while VM* only consists of a few *annotations* on models and diagrams. Thus, any modeler is, by definition, already capable of expressing simple queries, constraints, and transformations in VM*. For more complex expressions, a few new concepts have to be learned.

Environment transparency. Grace to syntax transparency, any modeling tool can be used as a front end for VM*. Therefore, any modeler can, by definition, use the VM* tool, which is just the editor the modeler uses anyway. Integrating the tool for executing a query or transformation, or checking a constraint, can be seamless.

Execution transparency. Unlike other approaches, VM* is not semantic, but syntactic, that is, it does not consider the meaning of model elements, but the notation alone. VM* is ignorant to what boxes and lines mean. While this imposes limits on the expressiveness of VM*, it also avoids semantic problems, makes the language more accessible, and simplifies implementation.

We argue that the restriction in expressiveness is rarely relevant in practice, while universal applicability, learnability, and usability are always a concern. VM* is applicable for any host language satisfying two conditions.

1. It must have a metamodel. This is trivially the case for any language that is implemented in a tool, in particular, for languages created with metamodeling tools like Adonis [15], EMF [23], or Meta-Edit [36].
2. It must have a way of adding textual comments to model elements, which is true for any modeling tool we have seen in practice.

It cannot be overemphasized that VM* is completely independent of the semantics of the host language. Conceptually, one may consider a model as a graph with labeled nodes. Unlike the original graph transformation approaches (e.g., [14, 18]), though, this graph is never exposed to the user. This means that VM* is applicable to many modeling languages, including languages for dynamic models like BPMN, EPCs, Use Case Maps, Simulink, or Role-Activity-Diagrams, as well as languages for static models like ER Diagrams, i*, KAOS, etc. Thus, VM* is also applicable to broad-spectrum languages with multiple notations, like UML, SysML, IDEF, or ArchiMate. Similarly, VM* is applicable to many *Domain Specific Languages* (DSLs). It would be exceedingly difficult to ascertain that VM* works with *any* visual notation as its host language, but we believe the prerequisites are very modest. We have yet to encounter a notation that cannot serve as a host to VM*. As a matter of notation, when referring to the instantiation of VM* for UML, we write VM*UML, and VM*BPMN for the instantiation of VM* for BPMN.

When using the same editor to create source models as well as the queries, constraints, and transformations to be applied to them, it is also irrelevant how models are represented in the editor. In practice, most modeling tools are less than completely compliant to whatever standards they aspire to implement. So, a model query

implementation that relies on standard compliance may be of limited use, or entirely incompatible. This way, many research prototypes are tied to the single modeling environment in which they happen to be implemented. It is hard to overstate the benefit of execution and environment transparency for industrial applications.

2 Examples

In this section, we present a high-level process model expressed as a UML Use Case diagram, a low-level process model expressed as a UML Activity Diagram, and another low-level process model expressed in BPMN. Throughout this paper, we denote metamodel concepts and VM* expressions by typewriter font and CamelCaps, while “elements” from sample models or queries are printed in sans-serif font and enclosed in quotation marks.

2.1 High-Level Process Models Expressed as Use Case Diagrams

As a first example, consider Fig. 2. Here, we use UML Use Case Diagrams to model the high-level views of process models, like function trees Value Added Chain Diagrams and process landscapes [13, pp. 239]. The Use Case Diagram at the top left represents the model repository; the other diagrams represent actions on the model: progressing clockwise from the top right, we see an example of a query, a constraint, and a transformation.

The query in Fig. 2 (top right) is a find pattern identified by the looking-glass icon. It matches all its elements against the model repository. For every successful binding of *all* query elements, one solution is generated. The wildcard acts as expected, matching any string. Therefore, the query will yield two results: the use case “request installment loan” and the use case “request revolving loan”. Use cases “specify loan details” and “buy credit insurance” do not match because their names do not match. Use cases “calculate risk” and “request loan” do not match because they are not associated with an actor. Use case “request loan” also does not match because it is abstract, and the use case of the query is not. In this query, only one annotation is required, namely the name pattern of the use case. Since this is a frequent case, we allow using wildcards in names without an explicit annotation. Escaping wildcards symbols allows to use them as proper symbols of names.

The constraint in Fig. 2 (bottom right) is again a find pattern, but this time it has a context annotation which defines the application condition of a constraint. Executing a constraint works just like executing a query: all the elements of the

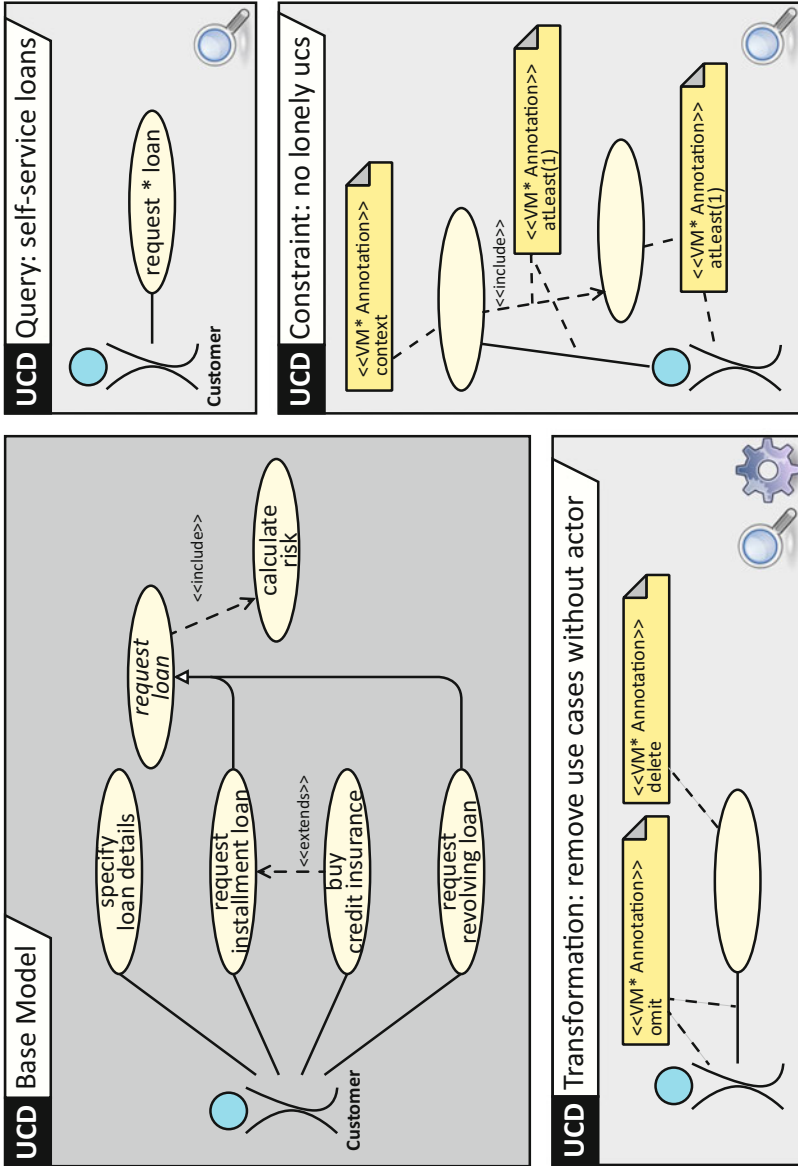


Fig. 2 An introductory example of a model repository (top left), a query, a constraint, and a transformation (clockwise from top right)

constraint are matched against the model repository. However, the results are treated slightly differently. There are three cases:

- If there is no binding for all of the elements of the context, the constraint is not applicable.
- If there is a binding for all context elements, but not for all the other elements, the constraint is violated, and the binding set is returned.
- Finally, if both the context elements and the other elements are matched, the constraint is satisfied.

The following schema summarizes when VM* constraints are applicable, and when they are satisfied or violated.

Elements matched		Constraint is
Within context	Outside context	
Not all	–	Not applicable
All	None	Violated
All	Some	Violated
All	All	Satisfied

In the interest of compact specification, this rule is modified by `atLeast` annotations. An individual `atLeast` annotation groups together related model elements. The annotation set relates such element groups, specifying how many times such groups must be matched to satisfy the constraint. In the example in Fig. 2 (bottom right), the parameter is 1, meaning that either there is at least one `Includes` relationship or one `association` attached to the “context” use case. The actor and the included use case are required because the UML syntax demands that `Associations` and `Includes` relationships may not be “dangling”. Connected to other model elements as they are in this example, however, they would have to match in order to not violate the constraint. Since the intention is for these elements not to match, we need to add another `atLeast` constraint. This constraint is satisfied for “Base Model”: the first four use cases are associated with “Customer”, “calculate risk” is included in another use case, and “request loan” is abstract, while the context use case in the constraint is not. Hence the constraint in Fig. 2 (bottom right) is satisfied.

Figure 2 (bottom left) shows a simple transformation for cleaning up “orphan” use cases. It consists of a single transformation rule expressed as one diagram defining both the left-hand side and the right-hand side of the rule. The left-hand side is a query for the model elements in the diagram, only that the actor and the association `prevent` matching due to the `omit` annotation. In other words, wherever these elements are present, the rule does not match. The right-hand side consists of the instruction to delete the use case.

In order to distinguish transformations from queries and constraints, yet express their similarity, the transformation is identified by both the looking-glass and the cogwheel icon. Transformations are interpreted similar to queries and constraints:

elements of the base model are matched against the elements of the transformations as explained above. Then, those annotations that indicate updates are triggered, in this case the `delete` clause. In this particular example, there are the following cases:

- If the use case is matched, and there is also an associated actor that matches, the whole transformation fails, and is not executed, because of the `omit` annotation on the actor and its association.
- On the other hand, if the use case is matched, but there is no associated actor, the transformation can be applied, deleting the “orphan” use case.

In this example, only “calculate risk” fits into that mold and is deleted. “calculate risk” fits structurally but is abstract.

2.2 Low-Level Process Models Expressed as Activity Diagrams

Now consider an example of a process model expressed as a UML Activity Diagram shown in Fig. 3. As before, there is one diagram that we use as the model repository (Fig. 3, left). To illustrate the VM*UML language capabilities, we use a query that we develop in several steps.

Suppose we want to find out what happens after a loan application is received and before the eligibility report is sent out, i.e., what are the exact steps to determine whether a client receives a loan or not? The general idea is to specify the delimiters “receive loan application” and “send eligibility report”, and find `Actions` between them. A first attempt to express this query may look like Q2a (Fig. 3, top right). Executing this query yields the empty result set, though, as we have specified that there should be a single `Action` that is *directly* connected to both delimiters. In the base model, however, there are *paths* of varying lengths.

In order for this query to find all `Actions` at *any* distance from the delimiters, we have to relax the condition and allow paths of arbitrary lengths instead of directly connected `Flows`. This is achieved by annotating the arcs with `steps = *`, which means “any number of steps” (see query Q2b, at the middle right of Fig. 3). However, this still yields no results. The reason is that the definition of `steps` restricts paths to contain only the kinds of node types adjacent to the `ControlFlow` arc on which it is defined. In this case, there is a plain `Action` (“send eligibility report” and the target node in the middle) and a `ReceiveEventAction` “receive loan application”, but no `ForkNode` or `JoinNode`. However, all the paths in the desired result set do contain fork or join nodes. So, in order to yield the expected result, we need to also relax the types of nodes on paths by using the `type` annotation in the next query (Q2c, at the bottom right of Fig. 3). Going beyond the example, we could be even more relaxed here and allow any type of node by specifying `type_is_any`.

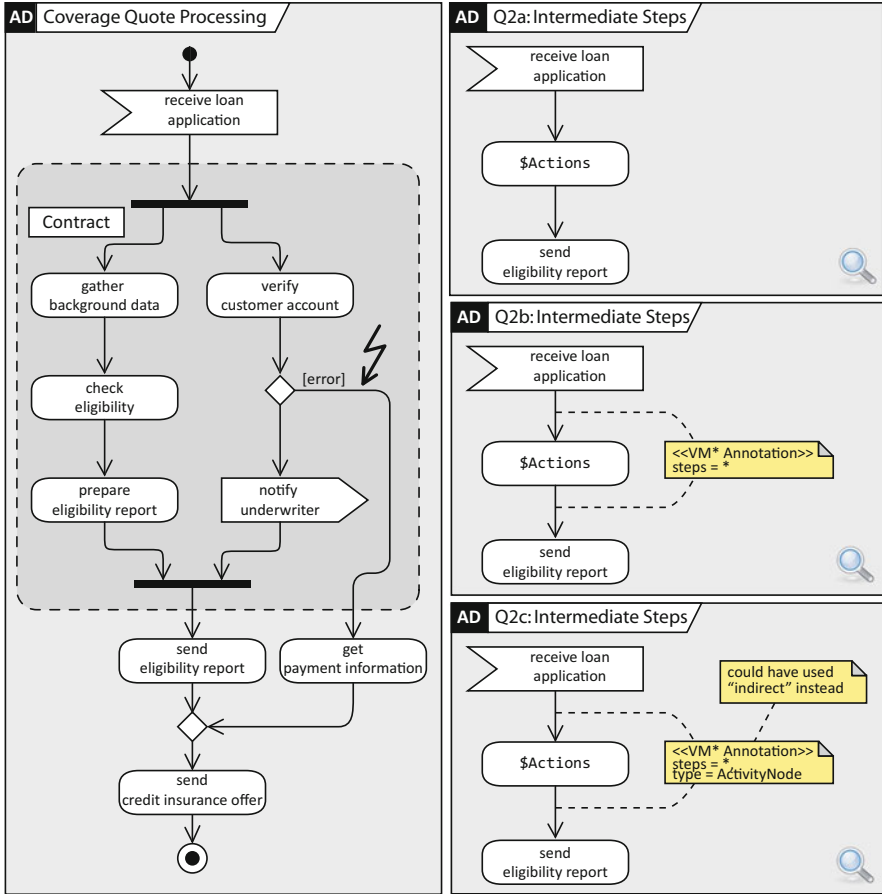


Fig. 3 A UML Activity Diagram with an exception (left) and related queries (right). Note that in UML, tokens within an InterruptibleActivityRegion (the gray area with a dashed borderline) are discarded upon firing an ExceptionFlow edge (marked with a lightning symbol)

2.3 Low-Level Process Models Expressed as BPMN Diagrams

In order to support our claim of general applicability, we now present VM* queries on *BPMN* (see [35] for more details). The top half of Fig. 4 defines how insurance quote requests are processed by an insurance company, while the bottom half presents five queries.

Suppose a business analyst is interested in finding all activities that deal with insurance coverage. She can succinctly express this request in VM* using Query 1, consisting of a task named \$A and a comment containing VM* annotations. The name of the task starts with a “\$” sign, indicating that it is a variable declaration. The name of any matching activity in the source model must be bound to \$A.

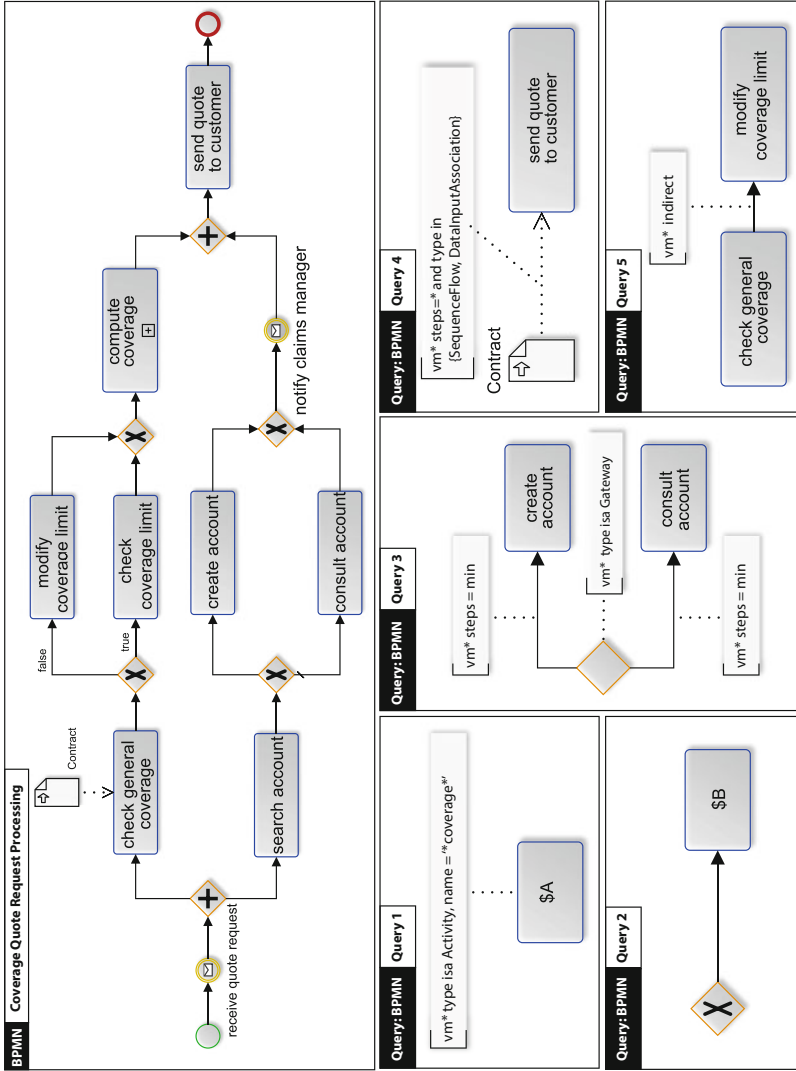


Fig. 4 Handling coverage quote requests in an insurance administration system: Process model expressed in BPMN (top), and five VM*BPIN queries (bottom)

The text annotation starts with the `vm*` keyword, indicating that it should be interpreted as a VM* expression. The annotation `type isa Activity` ensures that all of the BPMN activity types and their subclasses are considered, e.g., `Task`, `CallActivity`, `SubProcess`, and `SendTask`. In the example, this annotation enables the query to also return the *compute coverage* collapsed sub-process, which according to the BPMN metamodel is actually not a `Task`. Rather, `Task` and `SubProcess` are both subclasses of the `Activity` abstract meta class, see [19, p. 149]. Finally, the `name = '*coverage*` annotation uses a regular expression to specify that all activities matching the query must contain the string “coverage” in their name. A VM* implementation parses and evaluates such constraints when computing the result set. In the example, this applies to the first three nodes of the upper branch.

A similar, slightly more complex case is shown in Query 2. Here, the intention is to find all tasks which can only be executed after an `ExclusiveGateway`. This clearly applies to “modify coverage limit”, “check coverage limit”, and “create account”. Observe that it will also find “consult account”, even though the flow from the `ExclusiveGateway` to “consult account” is marked as default, while the flow in the query is not marked as default. The reason is that being a default flow is an optional property of the gateway, and the query does not specify a value for this property, which means it matches all values. On the other hand, “compute coverage” is not matched because its type is `SubProcess` while the type specified in the query is `Task`.

Query 3 detects if the “create account” and “consult account” tasks are executed exclusively, in parallel, or in some other manner, depending on the gateway preceding them. The `type isa Gateway` annotation indicates that the gateway preceding the tasks may be of any type as long as it is a subclass of the `Gateway` abstract meta class. The `default=any` is required to ensure that both default and non-default flows will be matched indiscriminately—this resembles the abstract property of UML `Classifiers`. The `steps = min` annotation limits the number of possible matches by stating that only the gateway-node closest to the two tasks should be returned. Considering the source model, Query 3 will then yield that the two tasks are executed exclusively.

The goal of Query 4 is to determine which part of the process has access to the “Contract” data input before a quote is sent to the customer. Observe that we allow paths of arbitrary length and all relevant types by the `steps` and `type` annotations, respectively. These two kinds of annotations occur frequently together so we have created the `indirect` annotation that is a shortcut for `steps=*` and `type is any`. So, Query 4 returns all paths from “Contract” to “send quote to customer”.

Query 5 illustrates the difference between syntactic and semantic querying. The intention of the query is to verify if the “modify coverage limit” task may be reached after executing the “check general coverage” task. Syntactically, the source model contains a path between the two tasks. Therefore, Query 5 will return this path. However, the question of reachability may not be reliably answered without considering semantics. Indeed, the path connecting the two tasks in the source model contains a false condition on one of its flow arrows, with the

intended meaning that control will never traverse this flow (i.e., task “modify coverage limit” will never execute). Answering this type of inquiry about process execution is a desirable but as of now unavailable feature in VM*. Also, it is this very feature of VM* that allows us to express Query 4 as succinctly as we have done, highlighting the trade-off between expressiveness and usability of VM*.

3 Query Language

In this section we discuss the abstract and concrete syntax, and formalize the semantics of VM*.

3.1 Abstract Syntax

VM* queries, constraints, and transformations consist of model fragments with (optional) textual annotations as defined above and formalized in the VM* meta-model, see Fig. 5. A VMStarExpression contains one or more Rules, each of which contains one or more Patterns, each of which contains one or more elements of the host language. All containments are exclusive, and there may be annotations on expressions, rules, or patterns. The components of the VM* metamodel must be mapped to existing elements of the host language metamodel. If

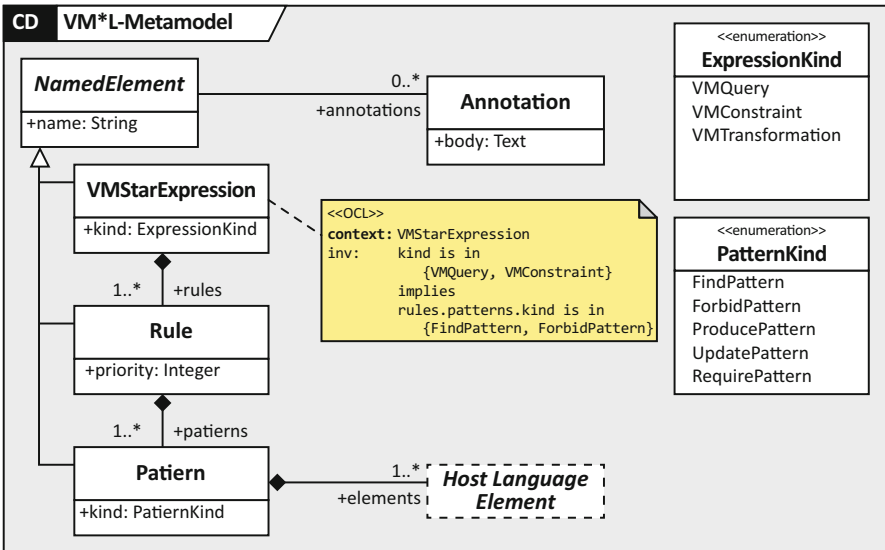







Fig. 5 The VM* metamodel

Table 2 Mapping of VM* metamodel constructs into UML language constructs to create VM*UML. In order to implement this mapping, a UML profile with these constructs must be created, and applied to any UML Packages containing VM* expressions

Stereotype	Applies to	Description
«VM* Annotation»	Comment	Annotation for any complete scope, e.g., a package of several expressions
«VM* Query» «VM* Constraint» «VM* Transformation»	Package	Annotation for a complete expression
«VM* Find» 	Package, Comment	Find patterns are used as queries, constraints, or as the <i>left-hand side</i> (LHS) of a transformation rule. A rule may contain at most one find pattern. A rule must contain either a find pattern or an update pattern
«VM* Forbid» 	Package, Comment	Forbid patterns are <i>negative application condition</i> (NAC) for a transformation rule or constraint. A rule can contain any number of forbid patterns and will be executed only if <i>none</i> of these patterns is matched in the source model
«VM* Require» 	Package, Comment	Require patterns represent a <i>positive application condition</i> (PAC) for a transformation rule. A rule can contain any number of require patterns, and will be executed only if <i>all</i> of these patterns are matched in the source model
«VM* Produce» 	Package, Comment	Produce patterns represent the <i>Right-hand side</i> (RHS) of a transformation rule, specifying how the target model is to be obtained from the source model. A rule may contain at most one produce pattern, and if there is a produce pattern, there must also be a find pattern in the same rule
«VM* Update» 	Package, Comment	Update patterns amalgamate a find and a produce pattern. There may be at most one update pattern per rule, and there may not be an update pattern and a find pattern together in a rule

the host language offers extension mechanisms, like the profiles and stereotypes of UML, these should be used. For instance, for VM*UML, annotations are encapsulated in UML comments annotated by the VM* Annotation stereotype. Table 2 shows how the VM* constructs map to the host language UML. For languages lacking extension facilities, naming conventions can be used to the same effect.

VM* specifies five pattern types: find, forbid, require, produce, and update. Queries and constraints consist of exactly one find pattern, and any number of forbid patterns. Constraints must have at least one annotation with the body context. In

transformations, all kinds of patterns may occur in any number. The transformation-annotations may only be used in update and produce patterns, and must be anchored to at least one model element of the pattern. Transformational patterns correspond to *Left-Hand Side* (LHS), *Right-hand side* (RHS), Negative, and Positive Application Condition (NAC and PAC, respectively) from graph transformation theory [14].

3.2 Concrete Syntax

The core of the VM* language is the set of annotations it provides. In the interest of expressiveness and succinctness, we sometimes cannot avoid referring to elements of the VM* metamodel explicitly. Also, sometimes execution options for queries, constraints, and transformations must be specified. Both of these can be achieved with annotations. See Table 3 for a complete overview of VM* annotations.

We start our overview of VM*'s annotation syntax by describing *user-defined variables*. They can be declared and manipulated within VM* annotations, and also used as meta-attribute values in pattern specifications. The names of user-defined variables are prefixed by the \$ character. Their *scope* extends across all patterns included in a query, and they are therefore employed for identifying corresponding model elements across different patterns. The type of a user-defined variable is inferred at query execution time. VM* supports the Boolean, Integer, Real, and String data types, in addition to the Element data type used for storing instances of host language meta classes. Regardless of their type, user-defined variables also accept the *undefined value* (“*”). A variable with this value is interpreted as possibly storing any accepted value of its respective data type.

For variable manipulation, VM* supports the arithmetic, comparison, and logic operators listed in Table 3. Logic operators can be expressed using shorthand notations (“,”, “;”, “!”, “->”) or full textual notations (and, or, not, if/then). The implication (“->”) and disjunction (“;”) operators can be combined to form a conditional if/then/else construct. The navigation operator (“.”) accesses model element meta-attributes, operations, and association-ends.

Apart from user-defined variables, VM* relies on *special variables* as a means of controlling query execution (the injective, precision and steps special variables) and accessing the contents of the source model (the id, self, and type special variables). Special variables have a predefined *scope*, identifying the specification fragment to which they are applicable. With the exception of the injective variable, the scope of all special variables is limited to the annotated model element. The injective variable has a global scope: its value determines how all patterns of a query are matched in the source model.

Clauses are the main building blocks of VM* annotations: each annotation consists of one or more clauses connected by logic operators. The use of clauses is inspired by logic programming languages and benefits annotation conciseness. A clause is an assertion about the pattern model elements to which it is anchored, about its containing pattern as a whole, or about user-defined or special variables.

Table 3 VM* annotations grouped by function (top to bottom): features for queries, constraints, and transformations, and generic arithmetic and logic operators

Annotation	Meaning
\$	Declares a variable to be matched (e.g., with the name of a model element)
match	Matches two variables (e.g., a name and a variable)
self	Denotes the model element to which an annotation is attached
id	Stores a model element identifier to match corresponding elements across patterns
?, *	The usual wildcards may be used in matching names of model elements
:=	Assigns a value to a user-defined variable, special variable, or model element meta-attribute
injective	When set to false, query elements may match more than one target model elements (default true)
steps	Used in an expression to restrict path lengths and types of nodes and edges on paths, allowing either comparisons to constants, the value * for “unrestricted”, or the values min or max denoting the paths extending to the shortest or longest paths available
indirect	Syntactic sugar for steps=* and type is any
type is x	Specify the type (meta class) of the annotated model element as being x, allowing any for x
type isa x	Specify the type (meta class) of the annotated model element as being a subclass of x
context	Anchor for constraints, must be present at least once in any constraint
omit	Annotated element must not be matched
atLeast (k)	At least k groups of annotated elements must be matched
atMost (k)	At most k groups of annotated elements must be matched
either	Syntactic sugar for atMost (1) and atLeast (1)
create	Part of the right-hand side of a transformation rule, create the annotated model element
create if not exists	Like create, but triggers only if the element does not already exist beforehand
delete	Part of the right-hand side of a transformation rule, delete the annotated model element
priority	Sets the priority of a rule (default 1)
+, -, *, /	The usual arithmetic operators. Note that the overloading of * can be resolved by context
=, <>, <, <=, >, >=	The usual comparison operators, where string comparison employs the wildcards * and ? in the canonical way
in	The usual set containment operator, where sets are enumerated between curly braces
like	A similarity-based matching operator for strings (global threshold)
precision	Annotated element is not required to match exactly but does allow a similarity-based matching with the given threshold
and, or, not	The usual logical operators
,, ;, !	Shorthands for logical and, or, and not
if <e> then <c1> else <c2>	The usual (eager) conditional, where the else-branch is optional, and -> is allowed as a syntactic shortcut
.	The usual dot-notation to access attributes of objects

The main role of clauses is to act as additional constraints on pattern matching. Note that variable assignment (“:=”) is treated as a clause.

The `either` clause can only be included in annotations anchored to several pattern model elements. All other clauses listed in Table 3 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 query-wide scope in VM*.

The annotations associated with VM* language elements play one of two roles: (i) When anchored to a host language element of a VM* pattern, annotations offer additional information or specify constraints related to that specific element. (ii) When anchored to the VM* expression itself, annotations specify execution options, such as global constraints on identifiers and variables.

3.3 Semantics

The central operation of the process of interpreting a VM* query, constraint or transformation is the *matching* of VM* patterns with corresponding source model fragments. For queries and constraints, the find patterns are matched against the source model, resulting in a set of intermediate bindings. Then, the forbid patterns are matched against the intermediate bindings, and the result is returned. For efficiency reasons, the annotations should be applied with decreasing strength. For instance, the `context` annotation is executed first.

For transformations, the rule with the highest priority is selected, and its left-hand side part is executed like a regular query. Any resulting bindings are then subjected to the right-hand side part of the rule. The rule application is continued until no more (new) results are yielded from executing the left-hand side. Then, the rule with the next highest priority is selected, and the process is repeated, until there are no more rules to apply. VM* currently only allows *endogenous* model-to-model transformations, that is, transformations in which the source and target models conform to the same metamodel [12, 16]. VM* transformations can be executed *in-place* to modify an existing model, as well as *out-of-place* to produce a new model.

As a means of formalizing the identification of matches between VM* patterns and a source model, it is useful to consider them both as typed attributed graphs. A *model graph* is defined as a typed attributed graph intended for representing models.

Definition 3.1 A *model graph* corresponding to a model M is a tuple $\langle N, E, T, A, V, type, source, target, slot, val \rangle$ where:

- N and E are finite sets of nodes and directed edges, respectively, with $E \cap N = \emptyset$;
- T is the set of node types corresponding to the meta classes included in M 's metamodel;

- A is the set of node and edge attributes corresponding to the meta-attributes included in M 's metamodel;
- V is the set of possible attribute values;
- $type : N \rightarrow T$ is a function assigning a type to each node;
- $source : E \rightarrow N$ is a function defining the source node of each edge;
- $target : E \rightarrow N$ is a function defining the target node of each edge;
- $slot : (N \cup E) \rightarrow 2^A$ is a function assigning a set of attributes to nodes and edges;
- $val : N \times A \rightrightarrows V$ is a partial function associating a value $v \in V$ to pairs (n, a) , where $n \in N$, $a \in A$, and $a \in slot(n)$.

Edges are uniquely defined by their source, target, and slots, i.e., $\forall e, e' \in E : (source(e) = source(e')) \wedge (target(e) = target(e')) \wedge (slot(e) = slot(e')) \implies e = e'$. The “undefined” element denoted \perp is not a member of any set.

The canonical subscript notation is used in what follows to denote elements of a particular model graph. For example, N_g and E_g denote the nodes and edges of model graph g . Both *bindings* and *matches* between a VM* pattern and a source model may be represented as model graphs; see Fig. 6 for an example representing Query Q1 from Fig. 2 as the actual query in the editor (top), the internal data structure a modeling tool might use to store the model (middle), and the semantic structure as a graph with labeled nodes (bottom).

Definition 3.2 Given two model graphs q and m representing a VM* pattern and a source model, respectively, a *binding* is an injective function $\beta : N_q \rightarrow N_m$ from the nodes of q to those of m .

Computing a binding generates potential matches, but actual matches must meet two more conditions. First, nodes mapped by the binding must have the same type. Second, model nodes must have at least the same slots, values, and interconnecting edges as the query nodes they are bound to.

Definition 3.3 A binding β is a *match* between a VM* pattern q and a source model m iff the following conditions hold:

- (i) $\forall n \in N_q : (type(n) = type(\beta(n))) \wedge \forall a \in slot(n) : val(n, a) = val(\beta(n), a)$,
- (ii) $\forall e \in E_q : \exists e' \in E_m : slot(e) = slot(e') \wedge \beta(source(e)) = source(e') \wedge \beta(target(e)) = target(e')$.

We define binding and match in two separate steps to highlight the algorithmic structure of VM*: Computing the binding generates potential solutions that are then pruned by computing the match. Implemented naively, this approach is computationally inefficient, and practically useless. Informing the binding-algorithm with the matching constraints, however, drastically reduces the complexity.

Most of the VM* annotations introduced in Sect. 3.2 have no other effect on the above definitions than to simply modify a pattern before it is matched with a source model. The `self`, `type`, and `steps` special variables are such examples. Other annotations such as `either` and `optional` imply that several different versions of a pattern must be matched with the source model. Again, this does

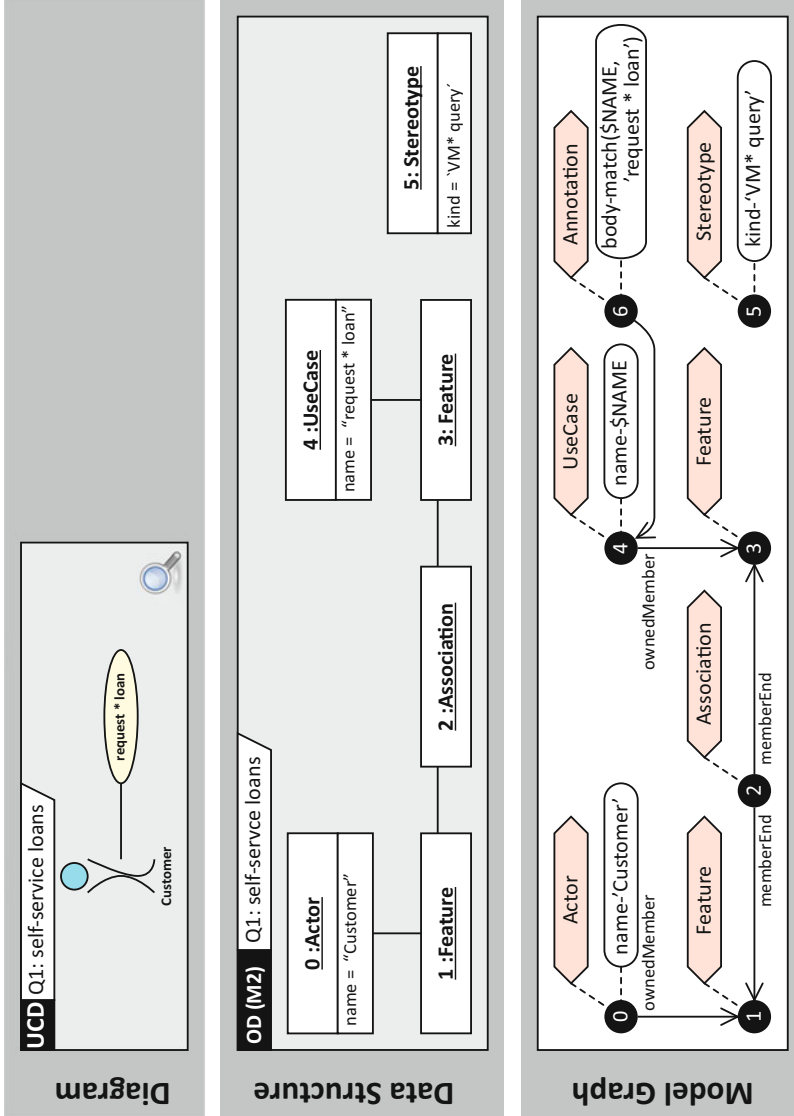


Fig. 6 Query Q1 from Fig. 2 at three different levels of abstraction: the actual query (top), the internal data structure a modeling tool might use to store the model (middle), and the semantic structure as a graph with labeled nodes (bottom)

not interfere with each individual pattern's matching process. The `unique` clause imposes a filter on match results after they have been computed, i.e., only one match is allowed. All of the aforementioned annotations do not interfere with the match computation, but simply work on the results. The only annotations affecting match computation are the `injective` and `precision` special variables: assigning `false` to `injective` removes the injectivity condition in Definition 3.2. Adjusting `precision` tweaks the matching precision.¹

4 Implementation

Developing VM* iterated through many cycles of conceptual work, exploratory prototyping, and evaluation not dissimilar to the design science methodology. In this way, three product lines have emerged over the years, implementing (parts of) VM* in turn.

moq We started with exploratory coding in PROLOG to determine the algorithmic feasibility and complexity [25, 26]. This branch later developed into query textual interfaces to study the concepts independent of the notation [30, 31]. The last step in this line is the MACH environment [33] which is available for download.² It can also be used without installation on SHARE [37].³

MQ We started exploring the visual notation aspect for model manipulation, with the **ModelQuery** systems, **MQ-1** [39] and **MQ-2** [3, 5].⁴ Both are plugins to the MagicDrawTM modeling environment.⁵ These implementations allowed to validate the overall approach, the syntax, semantic details, and the query execution performance under realistic conditions. A screenshot of **MQ-2** is shown below in Fig. 8.

vm* Finally, in order to prove the execution transparency of VM*, we realized it on two fundamentally different execution engines: Henshin [11], and as a REST-style Web-interface [4, 10].

Due to the limited space available, we can only explain one implementation here. We select **MQ-2**, since it is the basis of most evaluations. In the remainder of this

¹ Over the course of the years, the semantics of VM* languages has been defined in different terminologies and notations, and with slightly different meanings. The view presented here is the one proposed in [4], superseding earlier definitions such as the logic programming-based formalization presented in [29]. There, all annotations are viewed as logic constraints to be checked by an inference engine as part of the match computation process. Here, however, we decouple pattern matching from annotation interpretation, thus allowing a much wider array of existing matching engines, particularly ones based on graph matching.

² <https://www.pst.ifi.lmu.de/~stoerle/tools/mach.html>.

³ <http://fmt.cs.utwente.nl/redmine/projects/grabats/wiki>.

⁴ <https://www.pst.ifi.lmu.de/~stoerle/tools/mq2.html>.

⁵ www.magicdraw.com.

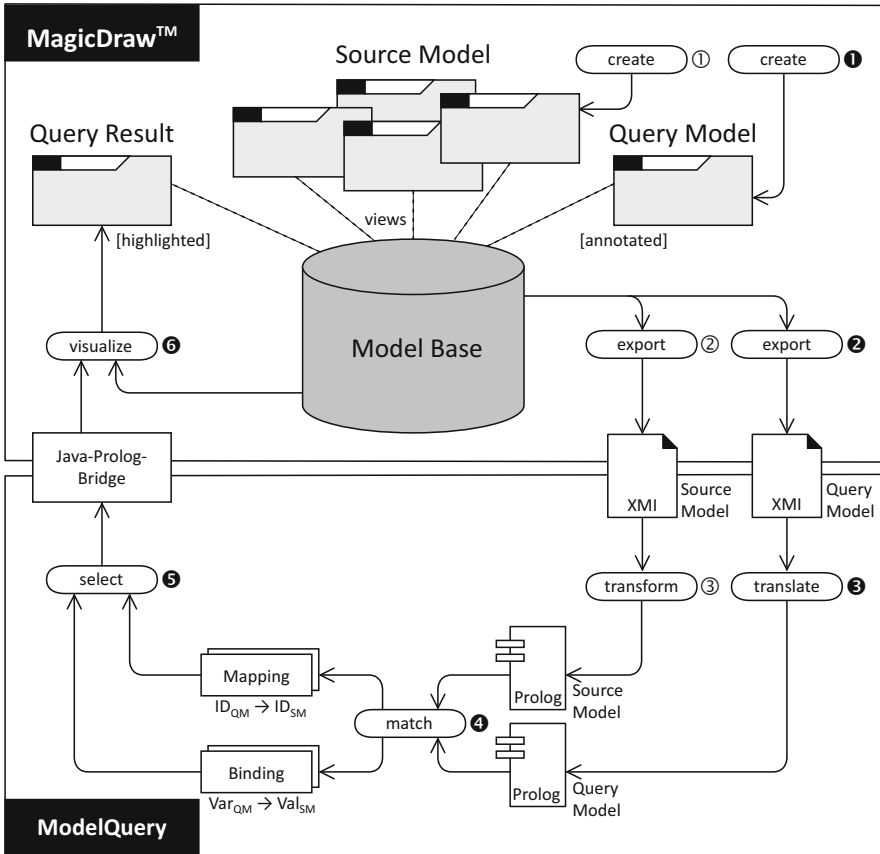


Fig. 7 The architecture of the **MQ-2** system and the main dataflow of executing queries. Numbers in black circles indicate the sequence of steps in creating and executing a query. Numbers in white circles indicate the steps for creating or changing models in the model base. Rectangles are used to represent data. Rectangles with rounded corners are used to represent actions. Arrows indicate dataflow

section, we will refer to VMQL rather than VM*, because VM* did not exist yet at the time of implementation. Figure 7 shows an architectural overview of **MQ-2**, while Fig. 8 presents a screenshot. The process of executing a query is shown by the numbers in black/white circles in Fig. 7. We will start with the white circles that highlight the steps for transforming the model base.

- ① A source model is created using some modeling language (UML in this implementation) and stored in the tool's model base.
- ② The model base is exported to an XMI-file, the standard file-representation of UML, using MagicDraw's built-in export facility.
- ③ The XMI-file is mapped into PROLOG predicates (see [29] for details). The mapping is bidirectional and generic, i.e., it does not limit generality of the

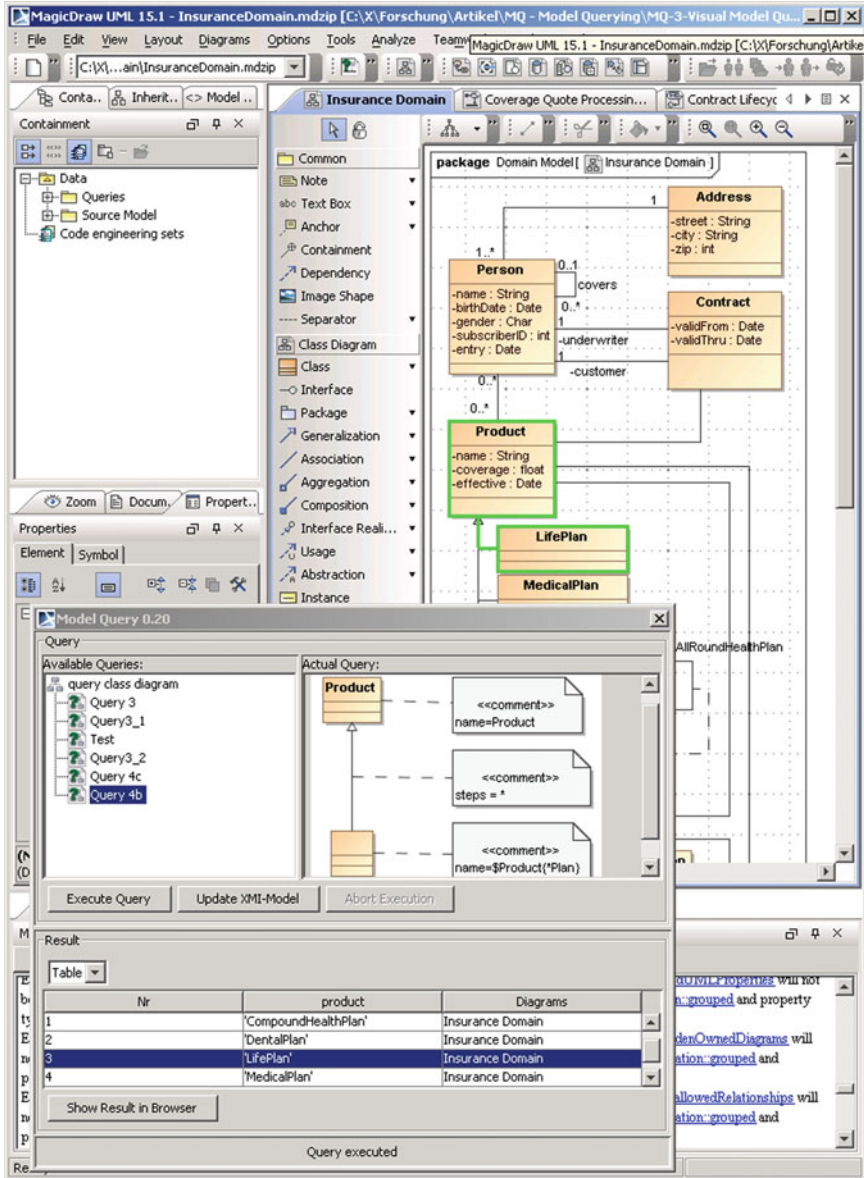


Fig. 8 The prototype implementation of VMQL. In the foreground, MQ-2 presents a list of “available queries” (top left), the one currently selected (top right), and the current result binding (bottom). One of the result bindings is selected, and a diagram in which this binding appears is shown in the background. The bound elements are highlighted with bold green borders

solution. Note that later implementations map models into graphs rather than to PROLOG predicates.

Now we turn to the process of translating and evaluating queries, indicated by numbers in black circles in Fig. 7. Since VMQL queries are annotated fragments of regular models, executing a query on a given model base boils down to finding matches between the query and the model base, and checking the constraints provided by the annotations.

- ① A VMQL expression is entered as a regular UML model with constraints packaged in stereotyped comments.
- ② ③ Then, the model query is transformed just like the source model. Constraints are directly mapped to predefined Prolog predicates and added to the predicates yielded from translating the query. Note that the precise way of how the constraint predicates are added is crucial for the computational complexity. In later implementations, queries are mapped to graph transformations instead of Prolog predicates.
- ④ Next, the predicate resulting from translating the model query is run on the Prolog-database resulting from transforming the source model. The two models are matched and the constraints are evaluated.
- ⑤ ⑥ Finally, the user selects one of the matches found in the previous step. To support this, a list of all diagrams containing elements of the match is computed. These may be either exact or approximate matches, as controlled by the `precision` constraint. If appropriate, the list is sorted by decreasing similarity. The diagram selected in the previous step is presented, and all elements of the binding are highlighted with fat green outlines (like in Fig. 8). The user may return to the previous step and select another match, and eventually terminate this query.

While the user interaction is tool specific, the query engine and the model interpretation are not (i.e., the lower part of Fig. 7). It should thus be fairly easy to port **MQ-2** to other UML tools, to DSL tools, or, in fact, to *any* modeling tool as long as it provides an open plugin API.

MQ-2 and the other implementations of (parts of) VM* have matured over several years of iterative refinement. Still, none of these implementations has reached the level of maturity and quality to compare to commercial products. However, our implementations do serve as evidence for several points. First, they prove feasibility of VM* in all its parts and aspects, and the soundness of the concepts and ideas behind VM*. Second, the implementations allowed us to get a better understanding of the computational complexity of executing VM*. This is particularly important, because executing VM* is ultimately based on sub-graph matching, a problem well-known to be computationally expensive in the worst case. So, it was not clear from the outset whether VM* would lead to a viable solution for practical situations. As it turns out, VM* is indeed viable for the intended use case, namely interactive queries, checks, and manipulations of models by modelers. However, it is likely not capable of supporting online processing of extremely large

model repositories or very complex and large sets of model transformations. It is neither suitable to aggregate trace data into process models (“process mining”). Due to lack of space, we cannot explore this aspect further in the present paper, and refer the reader to [6]. Third, we claim practical value of VM* particularly for usability. In fact, it is this aspect that led us to develop VM* in the first place, and it is here that we have placed the greatest emphasis of our work.

5 Usability Evaluation

Usability is rarely considered an important concern in many modeling communities.⁶ For us, it is the key to the success of any modeling related approach. Therefore, the cornerstone of the model querying research done in the context of VM* is, of course, the series of user studies to explore the usability of various model manipulation approaches. Table 4 lists the empirical studies we have conducted over the years to evaluate the relative usability of various languages for querying or transforming models. In Table 4, every line corresponds to a distinct study. Obviously, it is impossible to present all experimental results here in adequate detail, so we refer the reader to the original publications once more and restrict ourselves here to a discussion of the insights we obtained.⁷

Our initial hypothesis was that visual query languages should “obviously” be much better than textual languages. This turned out to be wrong in study 1 [30, 32]: All participants (senior practitioners) reached a perfect score on the OCL condition, even though they had never seen OCL before. In fact, they reached perfect scores under *all* treatments, whereas the student participants in study 0 had reached fairly low scores under each treatment. When asked, the participants in study 1 would explain that they had considered it to be some kind of pseudo-code, and simply executed it mentally, based on their intuition of the names of the operators and functions. Based on their extensive personal experience, they apparently understood the concepts, even if the syntax was new to them, and, as they asserted, less comprehensible than the syntax of the other languages tested. This gave rise to the hypothesis that there are at least two relevant factors to understanding model queries: the querying concepts (abstract syntax) and the querying notation (concrete syntax). We also speculated that participants’ scores in our tests should be less relevant than their cognitive load. Therefore, we started asking for preferences as a hint towards load levels.

⁶ Of late, some different opinions are heard, though, cf. e.g., [1].

⁷ We cite the main publications reporting on these studies, though many times results contributed to several publications, and there are several publications presented (parts of) the results. The names of the languages have evolved over time, we use here the name that the respective languages have had at the end of the research program, to make comparison easier for the reader. The first study was an exploratory pilot study to develop the research question rather than to provide meaningful results.

Table 4 Main empirical studies evaluating VM* and its precursors. In column “Method”, E, QE, and TA refer to **E**xperiments, **Q**uasi-**E**xperiments, and **T**hink **A**loud protocols, respectively. The columns under “Participants” detail the kind and number of participants in the study (**S**tudents, **P**ractitioners, and domain **E**xperts). In column “Mode”, R, and W stand for reading and writing of queries or transformations

No.	Ref.	Method	Participants			Languages	Intent	Mode
			S	P	E			
0	[26]	QE	5			VMQL, OCL, NLMQL, LQF	Query	R, W
1		QE,TA		5				
2	[30, 32]	E	12	6	6	VMQL, OQAPI, NLMQL	Query	R
3		E	16					
4	[29]	E	20			VMQL, OQAPI	Query	R, W
5		E	17					
6	[4]	E	24			VM*, OQAPI	Query	R, W
7	[4, 10]	E	30		4	VM*, Epsilon, Henshin	Transformation	R
8		E	44					
9		TA				VM*		

In studies 2 and 3, we switched from an exploratory research method to the classic experimental paradigm. Besides increasing the number of participants, we also explored sub-populations with different levels of qualification, and studied the controls more carefully, with surprising effects. NLMQL is a made-up purely textual model query language that we had introduced as a control, pitching textual vs. visual styles of querying. We had expected that the visual VMQL would outperform not only OCL (including the improved OCL-variant OQAPI) but also NLMQL. But the opposite happened, which led us to speculate that there are more factors at play than just the visual or textual notation of a query language. We attributed the difference to the proximity of the concepts in the query language and the concepts referred to in the experimental tasks. Put in another way: part of the usability of a model query language could be found in the appropriateness of the language concepts.

The subsequent studies 4 and 5 replicated the effects found previously and confirmed our speculations. Thus, we consider it an established fact that there are clear differences in usability of different languages, both with respect to reading and creating queries and constraints. Furthermore, the available evidence suggests that there is indeed a second factor which we call *language concept appropriateness* (LCA). Actually, the influence of LCA seems to be larger than whether a notation is visual or textual. We also stipulate that the effect can be masked by expertise and intellectual prowess, and so manifests itself mostly under stress (e.g., time pressure), and through variation across a population rather than through in vitro experiments.

Re-analyzing our data, we also hypothesize that there might be a third factor at play beyond the syntax and concepts of the queries that were presented to our participants: in all our experiments, the answer options for participants to choose from were given as prose. This might bias our results in favor of the textual,

prose-like notation (NLMQL, see studies 0–3 in Table 4). There is currently no experimental evidence to confirm this speculation, though.

After study 5, a major redesign of the language took place in the context of extending it to covering model transformations, yielding the VMTL-language. Of course, we also had to switch our controls from model query to model transformation languages (MTLs). We picked two best-in-class languages, namely the visual MTL Henshin, and the textual MTL Epsilon. In studies 7 & 8, we obtained similar results than we saw previously for model query languages. These studies showed no substantial advantage of one language over the other, and we currently do not understand why this is the case. In particular, we were surprised by the comparatively good results found for the “Epsilon” treatment. We hypothesized that this is due to the sampled population: Their educational background (CS graduate students) might introduce a bias in favor of Epsilon, which is very similar in style to common programming languages. This would be in line with our very first results, where the strong CS background of the study participants obviously had allowed them to cope with a language like OCL, despite its obvious usability deficiencies [26, 29, 30].

However, this is of course far from the “normal” situation, where domain experts typically do *not* have CS expertise—they have *domain* expertise, and probably some experience in reading models, and maybe even in creating models. Such users, we speculate, should have a much harder time coping with OCL, Epsilon and other languages created by computer scientists for computer scientists. Such users, we believe, outnumber CS experts by far, and they deal with models (in their domains) on a daily basis. They have no experience in programming or Model Driven Development technology, nor do they have an intrinsic motivation to use this technology.

Pursuing this idea, we conducted an observational study involving participants with and without a programming background (study 9). Interested in their opinion and thought processes rather than their scores, we switched to a think aloud protocol to find out how users understood the various languages. As expected, the presence of substantial programming experience lead to a completely different approach as compared to the non-programmers. Given that this is an interactive, observational study based on a very small set of participants, our results do not support strong interpretations and broad generalizations. On the other hand, our results likely generalize from queries to constraints and transformations, based on the structural similarities outlined in the first section of this paper. So, queries are the essential building blocks for all kinds of model manipulations, and likely the most used part, too.

6 Applications and Use Cases

In the Software Engineering domain, several types of models occur frequently, e.g., class or entity-relationship models, state machines, interaction models, and so on. Numerous powerful model manipulation languages and tools have been developed in the context of the MDE-paradigm, which places “*model transformation at the*

heart of software implementation” [22, p. 42]. Their origin has shaped them in a profound way, particularly considering the application scenarios, model types and the capabilities expected from modelers: Understanding and using MDE-style model manipulation approaches hinges on a perfect understanding of the metamodels underlying the modeling languages. Obviously, this is hardly part of the skill set of most domain experts. This means that MDE-flavored model manipulation languages, while expressive and supported by powerful tools, cater only for a very small audience of MDE-experts.⁸

On the other hand, the BPM community has more readily considered modelers which are experts in the domain modeled rather than technology. Thus, usability of model manipulation tools is a concern of much greater importance here. However, this domain typically considers only one type of models, namely process models. Additionally, existing approaches typically only deal with models of one particular notation. So, while a potentially much greater audience is addressed, a smaller set of models is covered. We believe that conceptual models are abundant today, created and used by knowledge workers without MDE expertise—think of organizational charts, shift plans, mechanical and electrical engineering models, Enterprise Architecture models, and chemical process schemas. These types of conceptual models have complex and long-lasting lifecycles. They are created, refactored, translated, and migrated in much the same ways as software models. End-users working with conceptual models are usually academically trained and highly skilled in their domains. We call them *End-User Modelers* (EUM), and they are at the focal point of our vision of model manipulation. We aspire high degrees of usability and learnability to cater for

EUMs, yet expressive and generic enough to cover their many application intents and modeling languages. This is an instance of the “*Process Querying Compromise*” [21, p. 12]. Pursuing this goal we are prepared to give up a degree of expressiveness.⁹ We win, however, a world of applications, as we shall illustrate in the remainder of this section.

Imagine a world where lawyers, mechanical engineers, accountants, and other domain experts have access to a language and tool that allows them to specify model transformations, queries, and constraints in a manner so close to their application domain and so simple to use that they can actually do it themselves. This would empower large numbers of knowledge workers to validate and update their models in a more efficient and less error-prone manner. The economic benefits would be hard to overstate. For instance, in an, as yet unpublished, interview study, a leading software architect from a major automotive supplier said about model querying: “... so I asked them: *guys, how long does this and that take ... then I took their hourly rate, the working hours, and so on, calculated how much we would save if*

⁸ This might have contributed to the hesitant industrial adoption of MDE [38].

⁹ In fact, some queries cannot be expressed in VM* [29], and some VM* queries are not computable [4, Sec. 6.3]. Also, there is a poor worst case performance of the underlying execution algorithm. We argue, however, that VM* still covers a large part of the *actual* application space.

we can only shave off 5 minutes a day. Those tens of millions [of €]... they didn't ask any more questions after that."

As a first example, consider a supplier of financial services. Over the last decade, substantial new legislation has been implemented to regulate financial markets. It is now widely accepted that "*a comprehensive understanding of business processes is crucial for an in-depth audit of a company's financial reporting and regulatory compliance*" [17]. A current trend in making this possible involves audits of the business processes, or, to be more precise, audits of the business processes *models*. Given the large number of processes and applicable regulations, companies are struggling to have fast and cost-effective audits. If auditors can create their own queries on these process models, they will be more effective in narrowing down items to check manually. Since this is already beneficial, imagine the added benefit of replacing non-compliant patterns of activity with compliant ones automatically and consistently.

As a second example, consider an Enterprise Architecture scenario focusing on compliance and change-management. Industrial installations with potential impact on safety and environment are subject to stringent regulation explicitly demanding up-to-date digital models of the installation so that, e.g., malfunctions resulting in environmental pollution can be traced back. Also, emergency response forces need full technical details of a plant, say, to effectively carry out their work if needed. Imagine an oil rig where a maintenance engineer discovers a burst pipe spilling oil and blocking an evacuation corridor on the rig. Even more importantly than repairing the damage is forwarding the information to all people and systems concerned. With models as the backbone of information management of the industrial installation, this amounts to the need for fast and accurate update to several interconnected models. While speed is of the essence, ensuring that all the right elements of the model are found and consistently updated globally is difficult, slow, and error-prone for simple editing or search/replace. On the other hand, pre-defining changes or back-up models is insufficient due to the unpredictability of changes and mitigating actions. Deferring the update to a back office loses vital context information and takes too long time. In such a situation, the maintenance crew should do the update on the spot.

7 VM* and PQF

Polyvany and colleagues have introduced the so-called *Process Querying Framework* (PQF) [21]), which "*aims to guide the development of process querying methods*", where "querying" includes all kinds of model manipulations in their terminology. Using PQF as a frame of references, we discuss VM* using the concepts and viewpoints defined there.

PQF consists of four parts with a number of activities ("active components") to be executed on models. Of the 16 activities defined explicitly in PQF, the following six are instantiated in VM*.

Formalize, Index, Cache. The process of compiling VM* specifications into executable Henshin transformations [4] or Prolog programs [3, 5] is fully automated. The translation of models includes manipulations that amount to indexing and caching (see steps ③ and ⑤ in Fig. 7).

Inspect. Matched model fragments are presented to users using the same notation and tool used to create the host model and the VM* query specification. The modeling tool used for creating the target models in the first place is also used to inspect the model.

Visualize. The concrete syntax of the host modeling language is used to visualize query results (see Fig. 8 for an example).

Filter. PQF's notion of model repository corresponds to a common (large) UML model, so that selecting sub-models by VM* queries amounts to filtering in PQF's understanding of the term. PQF only considers static reductions of the search space, though.

Other components of the PQF are not instantiated in VM*. PQF also defines a set of variation points ("design decisions") by which process query approaches may vary. VM* has the following stance on these design decisions.

Design Decision 1: Models. VM* aspires to be applicable to all model types, and to all (common) modeling languages. The restrictions to our aspirations are discussed below and are fairly limited. As long as a modeling language has a visual syntax, is implemented in a tool, and is defined using a metamodel (or, indeed, a meta-metamodel such as MOF or Ecore, [20, 23]), VM* is applicable.

Design Decision 2: Semantics. VM* is based on syntactic matching. The underlying semantics is not considered in this process and may only enter the picture in special cases. Considerations of the execution semantics such as fairness, termination, and finiteness are not relevant for VM*. Conversely, semantic differences along these dimensions cannot be expressed in VM* other than by additional, semantics-specific annotations.

Design Decision 3: Operation. Considering a CRUD context, the VM* languages address the *query intents* Read, Update, and Delete. The fourth intent (Create) is not supported directly in the sense that VM* is not able to create models from scratch, though it could be achieved by update rules with empty application conditions. The supported *query conditions* (i.e., VM* annotations) are designed with learnability and comprehensibility as priorities.

These decisions come at a price. First of all, the generality and usability are achieved, sometimes, at the expense of expressiveness: some queries cannot be expressed in VM* [29]. We have discussed the trade-off between language expressiveness, usability, and predictability of query results in [4, Section 8.2]. We argue that VM* still covers a large part of the *actual* application space.

Second, some VM* queries are not computable [4, Sec. 6.3], and there is a poor worst case performance of the underlying execution algorithm, which, in the end, amounts to graph matching. While, theoretically, this incurs an exponential run-time for the worst case, this case is rarely met in industrial applications. With suitable

optimizations and heuristics, a practical solution has been implemented, as we have shown, at least for the scenarios we have considered.

8 Conclusion

We present VM*, a family of languages for expressing queries, constraints, and transformations on models. Our focal point is the End-User Modeler, i.e., a domain expert working with models, but without programming background. We claim that our approach is feasible and usable and present evidence obtained through many iterations of implementation and improvement as well as a string of empirical user studies. We also claim that our approach is suitable to work with almost any commonly used modeling language, including DSLs. Since 2007, there have been over 20 publications on VM* and the steps leading up to it (see Table 1). An extensive discussion of the related work is provided in [4, Ch. 3].

The genuine contribution of this line of research is that it is the first to take usability into serious consideration for model manipulation languages. VM* is also comprehensive in the sense that it applies to all widely used visual modeling languages, covers many use cases, and provides many practical advantages, e.g., it readily adapts to any modeling environment. Pursuing our goal of combining usability with generality, we traded in a degree of expressiveness: some queries cannot be expressed in VM* [4, Sect. 8.2], and some VM* queries are not computable [4, Sect. 6.3]. Also, the theoretical worst case run-time of the execution algorithm of VM* is exponential.

Our line of research claims to provide a greater level of scientific certainty than its predecessors or competitors, as it has been (re-)implemented several times, and evaluated for usability and performance to a much greater extent than other approaches as of writing this. An obvious gap in our validation is the absence of a large scale, real life case study, i.e., an observational study in industry where a sufficiently large set of modelers uses VM* for an extended period of time on actual work items. Obviously, such a study would require a sufficiently well-developed tool. While we have created many tools over the years implementing (parts of) VM*, none of them has reached the level of maturity and product quality to compare to professional solutions.

This suggests two desirable avenues of progress: an industry-grade implementation and an observational case study in industry. Both of these will be very hard to achieve, and we consider them long term goals. In the nearer future, we plan to provide a structured literature review (SLR) of this field, and a cognitively informed theory of the usability factors for model querying.

References

1. Abrahão, S., Bordeleau, F., Cheng, B., Kokaly, S., Paige, R., Störrle, H., Whittle, J.: User experience for model-driven engineering: Challenges and future directions. In: ACM/IEEE 20th Intl. Conf. Model Driven Engineering Languages and Systems (MODELS), pp. 229–236. IEEE (2017)
2. Acreţoiaie, V.: The VM* Wiki. <https://vmstar.compute.dtu.dk/>
3. Acreţoiaie, V.: An implementation of VMQL. Master's thesis, DTU (2012)
4. Acreţoiaie, V.: Model manipulation for end-user modelers. Ph.D. thesis, Tech. Univ. Denmark, Depart. Appl. Math and Comp. Sci. (2016)
5. Acreţoiaie, V., Störrle, H.: MQ-2: A tool for prolog-based model querying. In: Proc. Eur. Conf. Modelling Foundations and Applications (ECMFA), LNCS, vol. 7349, pp. 328–331. Springer (2012)
6. Acreţoiaie, V., Störrle, H.: Efficient model querying with VMQL. In: Proc. 1st International Workshop on Combining Modelling with Search- and Example-Based Approaches (CMSEBA'14), CEUR Workshop Proceedings, vol. 1340, pp. 7–16 (2014)
7. Acreţoiaie, V., Störrle, H.: Hypersonic - model analysis as a service. In: Sauer, S., Wimmer, M., Genero, M., Qadeer, S. (eds.) Joint Proc. MODELS 2014 Poster Session and ACM Student Research Competition, vol. 1258, pp. 1–5. CEUR (2014). <http://ceur-ws.org/Vol-1258>
8. Acreţoiaie, V., Störrle, H.: Hypersonic: Model analysis and checking in the cloud. In: Kolovos, D., DiRuscio, D., Matragkas, N., De Lara, J., Rath, I., Tisi, M. (eds.) Proc. Ws. BIG MDE (2014)
9. Acreţoiaie, V., Störrle, H., Strüber, D.: Model transformation for end-user modelers with VMTL. In: tba (ed.) 19th Intl. Conf. Model Driven Engineering Languages and Systems (MoDELS'16), no. tba in LNCS, p. 305. Springer (2016)
10. Acreţoiaie, V., Störrle, H., Strüber, D.: VMTL: a language for end-user model transformation. *Softw. Syst. Model.* (2016). <https://doi.org/10.1007/s10270-016-0546-9>. <http://link.springer.com/article/10.1007/s10270-016-0546-9>
11. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, O. (eds.) 13th Intl. Conf. MoDELS, pp. 121–135. Springer (2010)
12. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Syst. J.* **45**(3), 621–645 (2006)
13. Davis, R.: *Business Process Modelling with ARIS – A Practical Guide*. Springer (2001)
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*. Springer (2006)
15. Junginger, S., Kühn, H., Strobl, R., Karagiannis, D.: DUMMY. *DUMMY* **42**(5), 392–401 (2000)
16. Mens, T., Van Gorp, P.: A taxonomy of model transformation. *El. Notes Theor. Comput. Sci.* **152**, 125–142 (2006)
17. Mueller-Wickop, N., Nüttgens, M.: Conceptual model of accounts: Closing the gap between financial statements and business process modeling. In: Fill, H.G., Karagiannis, D., Reimer, U. (eds.) Proc. Modellierung, pp. 208–223. Gesellschaft für Informatik (2014)
18. Nagl, M., Schürr, A.: A specification environment for graph grammars. In: Ehrig, H., Kreowski, G., Rozenberg, H. (eds.) Proc. 4th Intl. Ws. Graph-Grammars and Their Application to Computer Science, LNCS, vol. 532, pp. 599–609. Springer (1991)
19. OMG: *OMG Business Process Method and Notation (OMG BPMN, v2.0.2)*. Tech. rep., Object Management Group (2014). Last accessed from <https://www.omg.org/spec/BPMN/2.0.2/> at 2018-06-22
20. OMG: *Meta Object Facility MOF (version 2.5.1)*. Tech. rep., Object Management Group (2016). Available at www.omg.org

21. Polyvyanyy, A., Ouyanga, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://eprints.qut.edu.au/106408>
22. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. *IEEE Softw.* **20**(5), 42–45 (2003)
23. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional (2009)
24. Störrle, H.: MoMaT – A Lightweight Platform for Experimenting with Model Driven Development. Tech. Rep. 0504, Ludwig-Maximilians-Universität München (2005)
25. Störrle, H.: A PROLOG-based approach to representing and querying UML models. In: Cox, P., Fish, A., Howse, J. (eds.) *Intl. Ws. Vis. Lang. and Logic (VLL)*, CEUR-WS, vol. 274, pp. 71–84. CEUR (2007)
26. Störrle, H.: A logical model query interface. In: Cox, P., Fish, A., Howse, J. (eds.) *Intl. Ws. Visual Languages and Logic (VLL)*, vol. 510, pp. 18–36. CEUR (2009)
27. Störrle, H.: VMQL: A generic visual model query language. In: Erwig, M., DeLine, R., Minas, M. (eds.) *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 199–206. IEEE CS (2009)
28. Störrle, H.: Expressing model constraints visually with VMQL. In: *Proc. IEEE Symp. Visual Lang. and Human-Centric Computing (VL/HCC)*, pp. 195–202. IEEE CS (2011)
29. Störrle, H.: VMQL: A visual language for ad-hoc model querying. *J. Vis. Lang. Comput.* **22**(1) (2011)
30. Störrle, H.: Improving the usability of OCL as an ad-hoc model querying language. In: Cabot, J., Gogolla, M., Rath, I., Willink, E. (eds.) *Proc. Ws. Object Constraint Language (OCL)*, CEUR, vol. 1092, pp. 83–92. ACM (2013)
31. Störrle, H.: MOCQL: A declarative language for ad-hoc model querying. In: Van Gorp, P., Ritter, T., Rose, L.M. (eds.) *Eur. Conf. Proc. Modelling Foundations and Applications (ECMFA)*, no. 7949 in LNCS, pp. 3–19. Springer (2013)
32. Störrle, H.: UML model analysis and checking with MACH. In: van den Brand, M., Mens, K., Moreau, P.E., Vinju, J. (eds.) *4th Intl. Ws. Academic Software Development Tools and Techniques* (2013)
33. Störrle, H.: Cost-effective evolution of research prototypes into end-user tools: The MACH case study. *Sci. Comput. Programm.*, 47–60 (2015)
34. Störrle, H.: Cost-effective evolution of research prototypes into end-user tools: The MACH case study. In: Knoop, J., Zdun, U. (eds.) *Proc. Software Engineering (SE), Lecture Notes in Informatics (LNI)*, vol. 57. Gesellschaft für Informatik (GI) (2016)
35. Störrle, H., Acreţoiaie, V.: Querying business process models with VMQL. In: Roubtsova, E., Kindler, E., McNeile, A., Aksit, M. (eds.) *Proc. 5th ACM SIGCHI Ann. Intl. Ws. Behaviour Modelling – Foundations and Applications*. ACM (2013). [dl.acm.org/citation.cfm?id=2492437](https://doi.org/10.1145/2492437)
36. Tolvanen, J.P., Kelly, S.: MetaEdit+: defining and using integrated domain-specific modeling languages. In: *Proc. 24th ACM SIGPLAN Conf. Companion, Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 819–820. ACM (2009)
37. Van Gorp, P., Mazanek, S.: SHARE: a Web portal for creating and sharing executable research papers. *Proc. Comput. Sci.* **4**, 589–597 (2011). <https://doi.org/10.1016/j.procs.2011.04.062>
38. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial adoption of model-driven engineering: Are the tools really the problem? In: Moreira, A., and others (eds.) *16th Intl. Conf. MoDELS*, pp. 1–17. Springer (2013)
39. Winder, M.: MQ – Eine visuelle Query-Schnittstelle für Modelle (2009). In: German (MQ – A visual query-interface for models), Bachelor’s thesis, Innsbruck University