

# Formalization of QVT-Relations: OCL-based Static Semantics and Alloy-based Validation

Miguel Garcia

Institute for Software Systems (STS)  
Hamburg University of Technology (TUHH), 21073 Hamburg  
<http://www.sts.tu-harburg.de/~mi.garcia>

**Abstract:** The OMG QVT standard aims at consolidating and simplifying the model transformation landscape by offering three domain-specific languages (Relations, Operational Mappings, and Core) inspired in the declarative and imperative paradigms. We focus on QVT-Relations, which allows declaring a transformation as a set of relations that should hold between concrete models. The standard states the well-formedness rules for QVT-Relations in English. We provide instead an OCL formulation, thus formalizing the static semantics of the language and allowing checking Abstract Syntax Trees (ASTs) before transformations take place. Additionally, we express the dynamic semantics of QVT-Relations in Relational Logic. With that, symbolic analyses (input coverage, output well-formedness) become possible with the Alloy model-checker, allowing the design-time certification of transformations with more reliable techniques than testing. A case-study confirms the practical benefits of the approach.

## 1 Introduction

In recent years, EMOF + OCL has gained acceptance as the approach of choice to define the structure and well-formedness of software artifacts, including workflow notations (BPEL [Ake]) and query languages (JPQL [Gar06]), among others. This consolidation has motivated the need for expressing *model transformations* and *inter-model consistency checks* in a compact manner. Examples include: (a) high-level compilation (e.g., from BPMN into WebML [Bra06]); (b) refinement (e.g., spelling out the behaviors sketched in UML Activity Diagrams, by using Statecharts); and (c) round-tripping (e.g., between the textual notation of a BPEL process and its tree-based visualization).

In order to address the above scenarios (including the *bi-directional* evaluation of transformations) QVT-Relations [Obj07] was designed to encode input-output relationships as pattern-matching expressions guarded by preconditions. Current tools check the syntactic validity of transformations, as well as the *static semantics* of the language. Beyond that, a transformation should guarantee the well-formedness of its output, in order to avoid runtime exceptions that disrupt the operation of a model-driven toolchain (in terms of compiler technology, an analogy would be a Java compiler failing to produce valid bytecode for well-formed input).

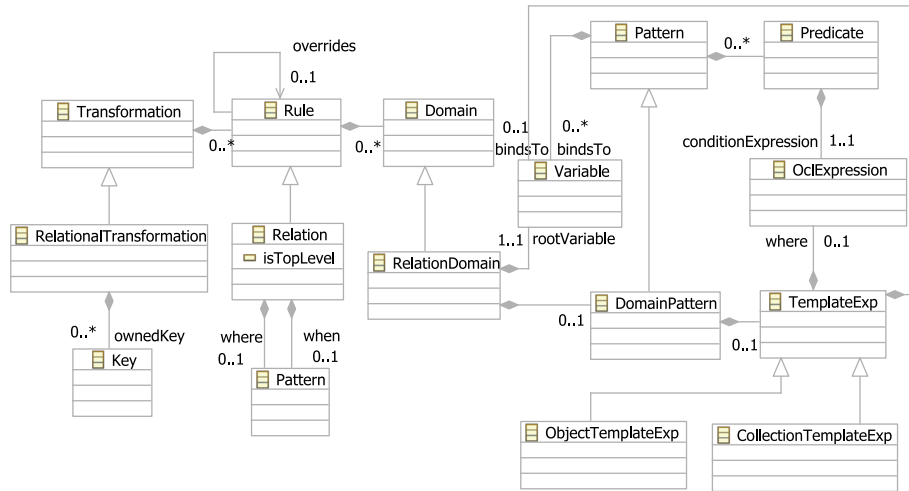


Figure 1: Fragment of the QVT-Relations metamodel

In order to perform such certification mechanically, we formalize the *dynamic semantics* of QVT-Relations using Alloy<sup>1</sup>, a logic engine developed at MIT by Daniel Jackson [Jac06]. The analyses thus made possible turn essential to debug transformations in an economic manner. For example, the case study in Sec. 5 uncovers a bug in the running example of the QVT standard, the UML2RDBMS transformation, where non-conformant output is produced for well-formed input.

Two capabilities of Alloy prove particularly useful: (a) the expressive power of its formalism, Relational Logic, which extends First-Order Logic with equality and transitive closure; and (b) visualization, which simplifies grasping the *counterexamples* found. An Eclipse-based plugin for Alloy is available<sup>2</sup>, although we intend to keep the operation of counterexample-finding transparent to the author of QVT transformations. Regarding the toolset used, screen captures were obtained from *medini QVT*<sup>3</sup>. Another QVT-Relations tool is *ModelMorf*<sup>4</sup>. Both are as of this writing free of charge for non-commercial use.

The structure of this paper is as follows. Sec. 2 reviews the main constructs of QVT-Relations, including the OCL-formulation of their Well-Formedness Rules (WFRs). After introducing Alloy in Sec. 3, the dynamic semantics of QVT-Relations are formalized in Sec. 4 and applied to a case study in Sec. 5. Related work is presented in Sec. 6, and Sec. 7 concludes. Knowledge is assumed about language metamodeling (in particular about EMOF + OCL as a mechanism to define Abstract Syntax). Familiarity with QVT-Relations is necessary to follow the discussion in Sec. 2 about its well-formedness rules.

<sup>1</sup>Alloy, <http://alloy.mit.edu/>

<sup>2</sup>Eclipse plugin for Alloy, <http://code.google.com/p/alloy4eclipse/>

<sup>3</sup>medini QVT, <http://projects.ikv.de/qvt>, open-source (EPL) since early 2008

<sup>4</sup>ModelMorf, <http://www.tcs-trddc.com/ModelMorf/index.htm>

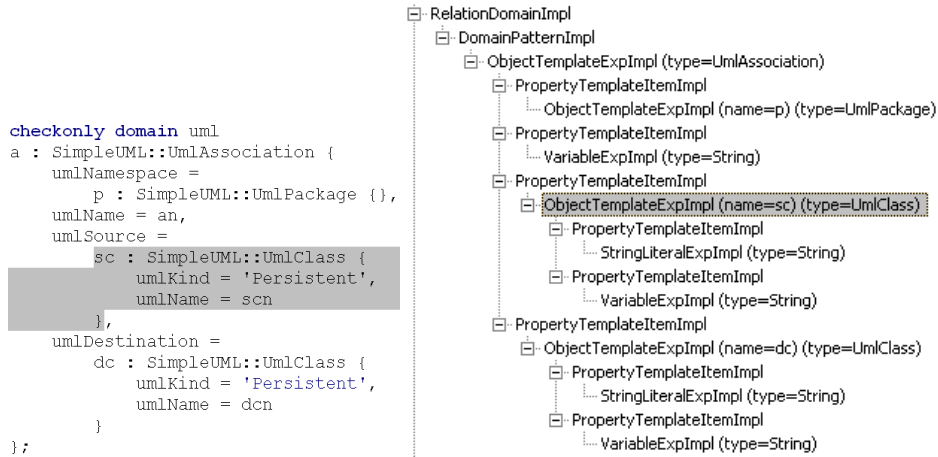


Figure 2: QVT text fragment and corresponding AST subtree

## 2 Static semantics, OCL formulation

A transformation manipulates classifiers (i.e., classes and datatypes) defined in one or more metamodels that the transformation expects as *parameters*. Thus, for example, a fully qualified path (rooted at a variable ranging over classifiers) comprises: a parameter name (different parameters may conform to the same metamodel), a package name, and a classifier name. More in general, the variables of a transformation range over *model elements*, e.g. “all the attributes owned by class *C* that have primitive type”. Although M0-level instances could also be manipulated, this usage is not anticipated by the spec.

Other QVT-Relations concepts are: *pattern matching*, involving *relation domains* (“domains” for short), *domain patterns* (“patterns”), *templates* (of *object* and *collection* kinds), and *variables*. For a transformation to be valid each of its top-level and transitively invoked relations should evaluate to true (which boils down to certain patterns matching). Referring to the metamodel fragment depicted in Figure 1, the AST nodes of relations and domains are easy to spot after their concrete syntax representation, which tags them with the keywords *relation* and *domain*. For the other AST nodes, no such mnemonic is available: curly braces may enclose the items in a template and the predicates in a pattern. Figure 2 compares an AST subtree with its textual serialization to showcase pattern nesting: variable *a* (an *UmlAssociation*) will bind to an instance whose fields have values as indicated, in particular those for *umlSource* and *umlDestination* being in turn patterns. The pattern for *umlSource* (highlighted in both the AST and textual views) introduces a variable for later use (*sc*). The two equalities required for a binding to succeed (*umlKind = 'Persistent'* and *umlName = scn*) are grouped by an object template (as opposed to a collection template) to denote that a single object will be matched. In the *SimpleUML* metamodel of the example, *umlSource* and *umlDestination* have multiplicity 1. In case any of the single instances reachable over them does not fulfill the equalities, the match for the enclosing template (for variable *a*) will fail as a whole.



Listing 2: Where do patterns go in the AST?

```

class DomainPattern extends QVTBase.Pattern {
  !ordered val QVTTemplate.TemplateExp templateExpression;
}
class Pattern extends EMOF.Element {
  !ordered ref EssentialOCL.Variable[*] bindsTo;
  !ordered val Predicate[*]#pattern predicate;
}
class Predicate extends EMOF.Element {
  !ordered val EssentialOCL.OclExpression[1] conditionExpression;
  !ordered transient ref Pattern[1]#predicate pattern;
}

```

Listing 3: What QVT patterns are made of

```

where { -- a Pattern owned by a Relation, owning Predicates
  ClassToKey(c,k);--Predicate with RelationCallExp as condition
  prefix = cn; -- condition is an OperationCallExp: equals()
}-- cn, a variable declared in the relation owning the where

```

Collection templates allow matching against a set, ordered set, bag, or sequence; using a notation inspired in that for function parameters in Haskell. For example, to match against a set-valued attribute, expected to contain at least three strings, one of which must be "ab", a preamble with three fillers is specified followed by the concat operator: `m:Set(String) { 'ab', s, _ ++ remainingElems }`. In general, the preamble lists a *fixed number* of elements, which may include, besides literal constants and nested templates, variables (`s` in the example) and the *underscore wildcard* that matches against anything (that we don't care to reference later). Different occurrences of `_` need not refer to the same element, unlike multiple occurrences of the same named variable. A new collection containing all of the original elements other than those matched by the preamble will be bound to the variable after `++` (`remainingElems` in the example, `_` would also have been allowed).

Collection templates over non-ordered collections are not the only potential source of combinatorial explosion. An object template may be used to match against each element in a collection, thus realizing cartesian products when several such templates appear side-by-side. For example, it is possible to obtain each combination (*attribute, operation*) declared in a class. Together with template nesting, these constructs render testing hopelessly non-exhaustive. The state-space reduction techniques of Alloy alleviate this problem.

Unlike the situation for a transformation, the spec does not include provisions for a relation to specify its list of expected arguments. The actual arguments for an invocation are required by tool vendors to match in order and type the relation domains. Therefore, as soon as a domain is added or otherwise the lexical order of domain declarations is changed, all existing invocations need to be refactored. Exchanging the declarations of two domains of the same type is a recipe for trouble: while existing invocations remain syntactically valid, the transformation will not behave as before (Listing 4).

Listing 4: Conformance between actual and formal args for invocations

```

context QVTRelation::RelationCallExp
inv actualFormalsConformance :
  argument→size() = referredRelation.domains→size() and
  argument→forall(arg | let i : Integer =
    self.argument→indexOf(arg) in
  assignmentCompatible(
    referredRelation.domain→at(i).eType, arg.eType)

```

### 3 Dynamic semantics, Alloy formulation

An Alloy specification [Jac06] declares concrete worlds, abstracted as set-theoretic relations that connect uninterpreted symbols (“atoms”). Each such concrete world (or snapshot of interest) is taken into account when performing analyses, which may be of three kinds:

(a) With *model finding*, (visual) depictions of all the finite concrete worlds that satisfy the specified constraints can be obtained (an analysis triggered with `run predicate`). Finding no concrete worlds is a strong indicator that the spec is inconsistent, i.e. all constraints cannot be satisfied simultaneously (however, some predicates could have been fulfilled in a larger finite *scope*).

(b) For unsatisfiable predicates, *Unsat Core* can be used to highlight the relevant portions of the Alloy spec that contributed to unsatisfiability.

(c) Assertions can be given, which are claimed to follow from the rest of the spec. *Counterexample finding*, triggered with `check assertion`, reveals finite concrete worlds that are conformant save for the broken assertion. *Regression testing* is similar in spirit to this analysis, only that not as systematic.

As to Alloy’s expressiveness about state evolutions over time, no dedicated syntax is provided (in contrast to imperative-style transformations [GM07]). Instead, Hoare-style pre and postconditions are preferred, where values in the pre and post snapshots are denoted by different variables, e.g.  $p$  and  $p'$ . The lack of dedicated Alloy syntax for EMOF’s String and numeric types has been pointed out as a disadvantage, as well as a missing counterpart for the OCL Standard Library, which is reused by QVT-Relations. These difficulties can be overcome by automating the encoding of the appropriate Alloy abstractions. For example<sup>6</sup>, if all a transformation  $T$  requires from Strings is comparison for equality, then the following Alloy definition will do: `sig String{}`. If, additionally, string comparison appears in  $T$ , then a more detailed abstraction (a linear ordering over the Character type) should be generated *by the encoding algorithm*. Together with the definition `sig String{content : seq Character}`, the OCL condition `a = b.concat(c)` can then be represented as in `sig T{ a, b, c : String }{ a.content=b.content.append[c.content]}`.

<sup>6</sup><http://tech.groups.yahoo.com/group/alloy-discuss/message/1266>

## 4 Methodology

Given that a transformation implicitly manages correspondences between model elements some notion of identity is necessary to warrant modification or deletion of the “right” counterpart from those available in the target model, or the creation of a genuine counterpart instead of a duplicate. QVT-Relations adopts the concept of *keys* from relational databases, in the form of per-class sets of fields. In terms of the Alloy formulation, keys are rephrased as constraints, and only duplicates-free models are considered during certification, thus contributing to scalability: the more constrained the input, the more pruning of the search space that a symbolic engine such as Alloy can perform early on (for a naïve generate-and-test methodology the opposite is the case, as significant effort is invested in the generation phase before the test phase can discard cases).

The basic steps to encode the definitions of OO classes and associations into some formalism are similar, as a comparison of the encodings into First-Order Logic [BKS02] and Relational Logic [ABGR07] shows: classes are formalized as sets of atoms (paying attention to disjointness and coverage) with additional predicates to capture the semantics of the specified relationships (inheritance being captured as subset, composition by ruling out multiple simultaneous owners as well as lack of owner, and multiplicities as restrictions on the number of links between atoms related by an association).

The encoding of QVT-Relations into Alloy starts with the *model type parameters* of a transformation  $T$ , i.e. the metamodels whose instances are passed as arguments to  $T$ . Given that different arguments may conform to the same model type, a separate population is required for each argument: pairwise disjoint subsets are defined. After that translation, Alloy can generate conforming arguments for  $T$ , to automatically explore the input space. In case the analyses should be limited to certain inputs, the pertinent definitions are declared *abstract* with a number of singletons populating them, thus preventing *model finding* from stipulating additional atoms. For `checkonly` domains a similar strategy is followed: without extra concrete definitions extending the abstract sets, the given populations are constant. Listing 5 depicts the Alloy declaration for class `ForeignKey`, with two WFRs for transformations to abide by.

At this point, functions can be encoded for (a) the templates in `checkonly` domains; and (b) `queries` defined alongside `relations` in  $T$ . Although such definitions may in turn involve pattern matching, the need to encode a particular search order is circumvented by letting Alloy bind variables to valid values: a predicate is specified parameterized with the variables and matching conditions that show up in the template of interest. Behind syntax, a set comprehension is at work. In Alloy,  $\{x_1:e_1, x_2:e_2, \dots, x_n:e_n \mid F\}$  denotes a set-theoretic relation with all tuples of the form  $(x_1, x_2, \dots, x_n)$  for which the constraint  $F$  holds, and where the value of  $x_i$  is drawn from the value of the bounding set expression  $e_i$ . The translation so far allows detecting whether some QVT-Relations variable is predestined never to be bound, as a result of the interplay between metamodel constraints and matching conditions. This kind of conclusion cannot be obtained with testing alone, unless prohibitively exhaustive.

Listing 5: Class encoded in Alloy with WFRs for transformation compliance

```

abstract sig ForeignKey extends RModelElement {
  fkColumns : some Column, -- at least one Column required
  refersTo : one Key
}{-- an FK in table T cannot contain columns not in T
all c:fkColumns | owningTable[this] = owningTable[c]
-- column types should match those of the PK being referred
fkColumns.type = refersTo.keyColumns.type
}

```

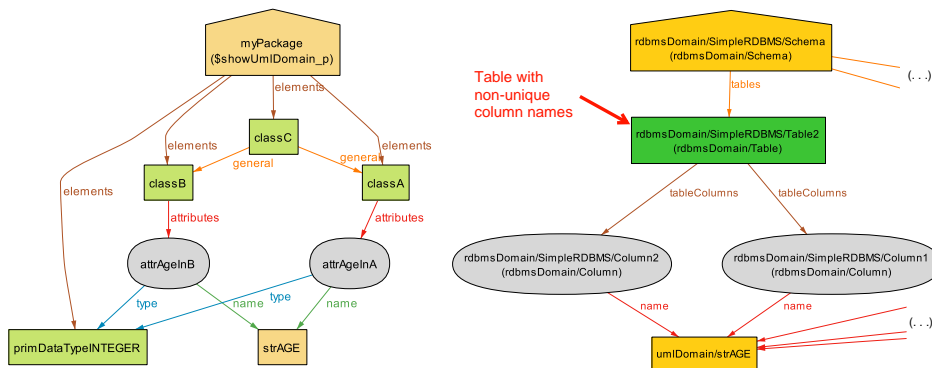


Figure 4: Counterexample for output well-formedness of UML2RDBMS

## 5 Case Study

In order to facilitate methodological comparison, we formalize the running example of the QVT spec (UML2RDBMS, transforming class models to relational database schemas, [Obj07, Annex A.1]) to later analyze whether well-formed output is obtained for each well-formed input. A counterexample is depicted in Figure 4, exhibiting duplicate column names in an output Table. The input shown left involves two superclasses declaring primitive attributes with the same name, which relation `SupperAttributeToColumn` maps unchanged. The transformation can be fixed by assigning a different attribute prefix for each such superclass (`SupperAttributeToColumn` had already the means to do this, but the prefix being passed was empty). The Alloy specs used to debug the transformation are available for download<sup>7</sup>.

For example, the relation `ClassToTable` is mapped to `all c:umlDomain/Class, t:rdBmsDomain/Table | guardClause[c,t] implies postCond[c,t]`. After identifying its essential features, the uncluttered visualization in Figure 4 was obtained. Subsequently, the transformation was run in a QVT-Relations tool for the reported input, obtaining the predicted malformed output but no warnings. The Alloy response times remained below one minute for most analyses performed.

<sup>7</sup><http://www.sts.tu-harburg.de/~mi.garcia/pubs/2008/qvtr/>

## 6 Related Work

Anastasakis developed a translation from UML class models and an OCL subset into Alloy [ABGR07], with details on OCL coverage provided in the tool documentation<sup>8</sup>. In follow-up work, a transformation between workflow notations was manually encoded in Alloy to certify output well-formedness [ABK07]. Dingel [DDZ08] uses Alloy to rectify the definition of the UML2 package merge operation.

Brucker and Wolff put forward HOL-OCL [BW06], a formalization of UML + OCL in the decidable logic HOL, to support formal validation. In contrast to model checking, which is always bound by a finite scope, proofs may be made about infinite models. The downside is the expertise required about proof tactics and the encoding of the problem domain.

## 7 Conclusions and Future Work

The notation in Annex B of the QVT spec is not suitable as semantic anchor (unlike Relational Logic) because, although it exhibits a resemblance to First-Order Logic, it cannot be mechanically checked.

The UML2RDBMS transformation does not aim at encoding all the rules required for realistic Object-Relational Mapping<sup>9</sup> (for example, no provision is made to handle many-to-many associations). Therefore, a minimal criteria was adopted in Sec. 5 (output well-formedness) as it suffices to demonstrate the methodology. An industrial-strength transformation should additionally be certified to preserve the “semantics” in terms of the languages for input and output (thus guaranteeing *compiler correctness* [Goe00]). In the example, we do not check whether the resulting database schemas have the same expressive power as the input class models. As it turns out they don't: although foreign keys are generated for many-to-one associations, such foreign keys are not included in the tables for subclasses, unlike the columns for inherited attributes. QVT-Relations cannot be expected to be aware about the semantics of all the languages whose sentences it may transform. As a consequence, compiler correctness has to be validated separately, by manually specifying verification conditions that cannot be extracted mechanically from the transformation under study.

We aim at automating the translation from QVT-Relations into Alloy, as well as the subsequent analyses (input coverage, output well-formedness) by building upon our Eclipse-based OCL compiler [GS07] and the EMF metamodel of QVT-Relations. Given the progress experienced by logic engines in terms of efficiency, portability, and componentization, we believe the time is right for adopting the best practice of anchoring the semantics of software modeling standards in decidable formalisms, as demonstrated in this paper for QVT-Relations.

---

<sup>8</sup>UML2Alloy, <http://www.cs.bham.ac.uk/~bxb/UML2Alloy/index.php>

<sup>9</sup>Scott Ambler on ORM, <http://www.agiledata.org/essays/mappingObjects.html>

## References

- [ABGR07] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In *Proc. of MoDELS 2007*, pages 436–450, 2007.
- [ABK07] Kyriakos Anastasakis, Behzad Bordbarand, and Jochen M. Küster. Analysis of Model Transformations via Alloy. In *4th MoDeVva Workshop*, pages 47–56, In conjunction with MoDELS07, Nashville, TN, USA, 2007. <http://www.modeva.org/2007/modevva07.pdf>.
- [Ake] D. H. Akehurst. Validating BPEL Specifications using OCL. Technical Report 15-04, Univ. of Kent at Canterbury. <http://www.cs.kent.ac.uk/pubs/2004/2027>.
- [BKS02] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the Object Constraint Language into First-order Predicate Logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark, 2002*. <http://www.uni-koblenz.de/%7Ebeckert/pub/verify2002.pdf>.
- [Bra06] Marco Brambilla. Generation of WebML web application models from business process specifications. In *ICWE '06*, pages 85–86, New York, NY, USA, 2006. ACM.
- [BW06] Achim D. Brucker and Burkhard Wolff. The HOL-OCL Book. Tech Rep 525. ETH Zürich, 2006.
- [DDZ08] Jürgen Dingel, Zinovy Diskin, and Alanna Zito. Understanding and improving UML package merge. *Software and Systems Modeling*, 2008. To appear.
- [Gar06] Miguel Garcia. Formalizing the Well-formedness Rules of EJB3QL in UML + OCL. In T. Kühne, editor, *Reports and Revised Selected Papers, Workshops and Symposia at MoDELS 2006, Genoa, Italy*, LNCS 4364, pages 66–75. Springer-Verlag, 2006.
- [GM07] Miguel Garcia and Ralf Möller. Certification of Transformation Algorithms in Model-Driven Software Development. In Wolf-Gideon Bleek, Jorg Räsch, and Heinz Züllighoven, editors, *Software Engineering 2007*, volume 105 of *GI-Edition LNI*, pages 107–118, 2007.
- [Goe00] Wolfgang Goerigk. Compiler verification revisited. In J Strother Moore, Matt Kaufmann, and Panagiotis Manolios, editors, *ACL2 case studies*, pages 247–264. Kluwer Academic Publishers, 2000. ISBN 0-7923-7849-0.
- [GS07] Miguel Garcia and A. Jibrán Shidqie. OCL Compiler for EMF. In *Eclipse Modeling Symposium at Eclipse Summit Europe 2007, Stuttgart, Germany, 2007*. <http://www.sts.tu-harburg.de/~mi.garcia/pubs/2007/ese/oclcompiler.pdf>.
- [Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. ISBN 0-262-10114-9.
- [Obj07] Object Management Group. MOF QVT Final Adopted Specification, formal/07-07-07, July 2007. <http://www.omg.org/docs/ptc/07-07-07.pdf>.