Journal of Computing and Security

2020, Volume 7, Number 1 (pp. 35–62)

http://www.jcomsec.org

Formalizing the main characteristics of QVT-based model transformation languages

Alireza Rouhi [a,*] Kevin Lano [b]

[a] *Faculty of Information Technology and Computer Engineering, Azarbaijan Shahid Madani University, Tabriz, Iran.*
[b] *Department of Informatics, King's College London, London, UK.*

ARTICLE INFO.

Article history:
Received: 22 February 2020
Revised: 22 May 2020
Accepted: 1 June 2020
Published Online: 19 June 2020

Keywords:
Model-Driven Development (MDD), Model Transformation, QVTr, Formal Model, Z Notation.

ABSTRACT

Model-Driven Development (MDD) aims at developing software more productively by using models as the main artifacts. Here, the models with high abstraction levels must be transformed into lower levels and finally executable models, i.e., source code. As a result, model transformation languages/tools play a main role on realizing the MDD goal. The Object-Management Group (OMG) presented the Query/View/Transformation (QVT) as a standard for the Meta-Object Facility (MOF)-based model transformation languages. However, implementing a model transformation language, which supports the full features of the QVT proposal requires a formal model of the underlying concepts. Having common terminology and a formal, precise, and consistent specification facilitates developing dependable transformation languages/tools. This paper aims to provide a formal specification of the main characteristics of a QVT-Relations (QVTr) model transformation language using the Z notation. The proposed formal model can be adapted for formalizing other domain and language concepts too. To show the applicability of the proposed formalism, a simplified version of the classic object-relational transformation is specified. Additionally, we show how the semantics clarifies some outstanding semantic issues in QVTr. The proposed formalism of this paper will pave the way to building support tools for model transformations in a unified manner in MDD.

© 2020 JComSec. All rights reserved.

1 Introduction

Model-Driven Development (MDD) intends to develop software more productively by applying models as the main artifacts [1–3]. A model which represents an abstraction of a real system must be transformed into a lower level using model transformations and finally yielding the implementation code, i.e., the executable model [4].

To realize MDD, Object-Management Group (OMG) has presented the Model-Driven Architecture (MDA) paradigm [5]. MDA which can be considered formally as a subset of MDD separates models into Computation Independent Models (CIMs), Platform Independent Models (PIMs), and Platform Specific Models (PSMs). The reason of this classification is the existence of a constant evolution in the implementation technology that requires software portability from one technology to another in the future. It

* Corresponding author.
Email addresses: rouhi@azaruniv.ac.ir (A. Rouhi), kevin.lano@kcl.ac.uk (K. Lano)
https://dx.doi.org/10.22108/jcs.2020.121740.1047
ISSN: 2322-4460 © 2020 JComSec. All rights reserved.

is worthy to note that the *transformation* operation is the main way of modifying and creating models in the MDA paradigm. The significant role of model transformation motivated OMG to define a standard language in alignment with its other standards. This effort led to the introduction of Meta-Object Facility (MOF) 2.0 Query/View/Transformation (QVT) language suite [6].

The QVT standard/language is composed of three constituent languages: QVTr, QVT-Core (QVTc), and QVT-Operational (QVTo). By exploring the literature, several languages and the related tools are found which aim at implementing the QVT language, e.g., mediniQVT [7], QVTr-XSLT [8], QVTo-Eclipse [9], SmartQVT [10] to name a few. Even though the mentioned tools claim their implementation basis is the QVT specification, there are considerable distinctions regarding their supported features [11]. Undoubtedly, lack of a formal model foundation is one of the main reasons that causes differences on their supported features [12]. Moreover, nearly half of the implemented tools based on the QVT standard have been discontinued [11].

Additionally, the descriptions provided by the QVT standard are semiformal, which yield ambiguities in the analysis and tool implementations [13], e.g. checking the conformance of models in the transformation specifications [6]. In particular, the QVTr semantics [6] is omissive with regard to the semantics of relation *when* and *where* clauses, and does not distinguish the cases of top relation and non-top relation execution.

To address the mentioned formal incompleteness of the QVT standard, some researches have been conducted regarding the formal characteristics of a model transformation language [14–18]. Exploring the literature reveals some related works [19, 20] which use a translation to modal μ-calculus and game theory in order to investigate the execution modes of a QVTr transformation in detail, i.e., checkonly/enforcement. Stevens [19], Bradfield and Walukiewicz [20] have proposed a formal semantics using μ-calculus to resolve some incompleteness and ambiguities regarding the QVT standard specifications. Due to the many similarities of QVTc and Triple-Graph Grammars (TGGs) transformations, Greenyer and Kindler [15] developed an interpreter which executes QVTc mappings by transforming them to the TGG rules. Guerra and de Lara [13] to fill the gap between the QVTr transformations and the underlying QVTc and QVTo operational counterparts presented a formal semantic for the QVTr check-only mode transformations based on algebraic specification and category theory.

n a simple statement, to clarify the ambiguities of the existing model transformation standards such as QVTr [6] and facilitate the development of supporting tools in MDD, it is required to formalise a common understanding of the model transformation terminologies.

The aforementioned approaches (1) have used formal semantics and notation [13, 15, 19] that are more complex and specialised than the Z notation [21] we use in this paper, which is based closely on classical logic and set theory, and (2) our formalism is relatively complete since it includes nearly all of the concepts which are required to specify a model transformation language standard. In particular, the underlying model and metamodel theory are formalised, together with the execution semantics of rules and of complete transformations. Thus, in this paper, we present a simplified and classical logic formalism abstracted from implementation concerns of concepts related to a typical MOF-based model transformation language using Z notation. As a result, the main contribution of this paper is proposing a formal model for the main charachteristics of the QVTr model transformation language with the following secondery outcomes:

- To pave the way for understanding of the common model transformation concepts, in order to provide a foundation for developing model transformation tools.
- To provide contributions to address some open issues in QVTr semantics.
- To clarify alternative semantic choices for QVTr constructs.

To show the applicability of the proposed formal model in practice, the specification of a simplified classic object-relational transformation [6, 12] is presented.

The paper is organized as follows. In Section 2, a brief introduction of the QVT standard is presented, the specification of OMG's proposed model transformation language in the MDA paradigm. Section 3 presents the formal model of the main model transformation concepts in the Z notation, together with applications of the semantics. Section 4 presents and discusses the related works focusing on the similar existing formal models. Finally, Section 5 concludes the paper and identifies future works.

2 Background

Transformation technologies are not new in the software engineering field [3]. A typical compiler of a programming language like C++, is an example of a transformer which receives a program source code, an artifact with a high level of abstraction and converts it to an executable code, an artifact in the lowest ab-

straction level. As another example, *XML* [22] which is one of the standard forms of data exchange, has in eXtensible Stylesheet Language for Transformations (XSLT) [8] a standard transformation language for Extensible Markup Language (XML) documents.

As stated previously, due to the important role of model transformation in MDA, OMG has specified a standard in alignment with its other standards named QVT [6].

2.1 The QVT Standard

The QVT specification has a hybrid declarative/imperative nature in which the declarative part per se is divided into a two-level architecture (Figure 1):

- A user-friendly language/metamodel and a declarative specification of the MOF model elements' relationships named *Relations* to support complex object matching and element creation via object templates. The traces and their instance classes induced as each transformation are created implicitly to keep record of what has been done during the transformation execution.
- A language/metamodel named *Core* which has been defined with a minimum extension to Essential MOF (EMOF) and Object Constraint Language (OCL). It supports pattern matching only on a flat set of variables by evaluating the pattern conditions against a set of models. The elements of source, target, and trace models are considered symmetric. The trace classes must be defined explicitly as MOF models. Moreover, the creation and deletion of an instance trace is done like the other model elements.

The two mechanisms for implementing the imperative parts of the *Relations* and *Core* languages are *Operational Mappings* and *Black Box Implementations*, which extend the imperative capabilities of the QVT standard. The transformations are unidirectional, and the trace models are implicit. In other words, to support bi-directionality, the transformations must be defined in both directions [6]. Of course, it must be mentioned that there are other tools in the model transformation literature in general [23] and the bidirectional model transformations in particular [12, 24, 25]. However, the focus of this paper is the QVTr based model transformation languages/tools.

2.2 The QVTr Model Transformation Language

In this language, each transformation defined as a set of relations between candidate models of the underlying transformation. Relations define the constraints that must be satisfied by the candidate model elements. The candidate models are named and must conform to some model types, i.e., their corresponding metamodels. In other words, the candidate model constituent elements are restricted to those element types defined on the referenced packages of their metamodels. A transformation can be invoked to check consistency between candidate models or to modify models for enforcing consistency among them.

```
transformation UML2RDBMS(uml: UML, rdbms: RDBMS) {
top relation PackageToSchema {
  n: String;
  checkonly domain uml p: Package { name = n };
    enforce domain rdbms s: Schema { name = n };
}

top relation ClassToTable {
  n: String;
  checkonly domain uml c:Class {
      persistent = true,
      namespace = p: Package,
      name = n
    };
  enforce domain rdbms t: Table {
      schema = s: Schema,
      name = n
    };
  when {PackageToSchema(p, s);}
  where {AttributeToColumn(c, t);}
}

relation AttributeToColumn {
  checkonly domain uml c: Class {};
  enforce domain rdbms t: Table {};
  where {
      PrimitiveAttributeToColumn(c, t);
      SuperAttributeToColumn(c, t);
    }
}

relation PrimitiveAttributeToColumn {
  n: String;
  checkonly domain uml c: Class {
     attribute = a: Attribute { name = n }
  };
  enforce domain rdbms t:Table {
      column = cl: Column { name = n }
    };
}

relation SuperAttributeToColumn {
  checkonly domain uml c:Class {
    superclass = g: Class {}
  };
  enforce domain rdbms t: Table {};
  where {
      AttributeToColumn(g, t);
    }
  }
}
```

Listing 1: Simplified Version of UML to RDBMS Model Transformation Specification in QVTr

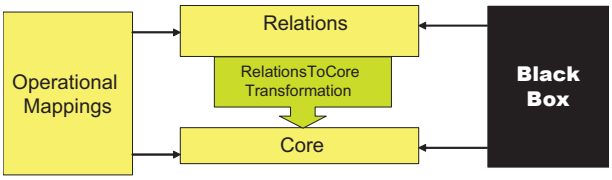

Figure 1. The Relationships of Constituent Metamodels in the QVT Model Transformation Language Suite [6].

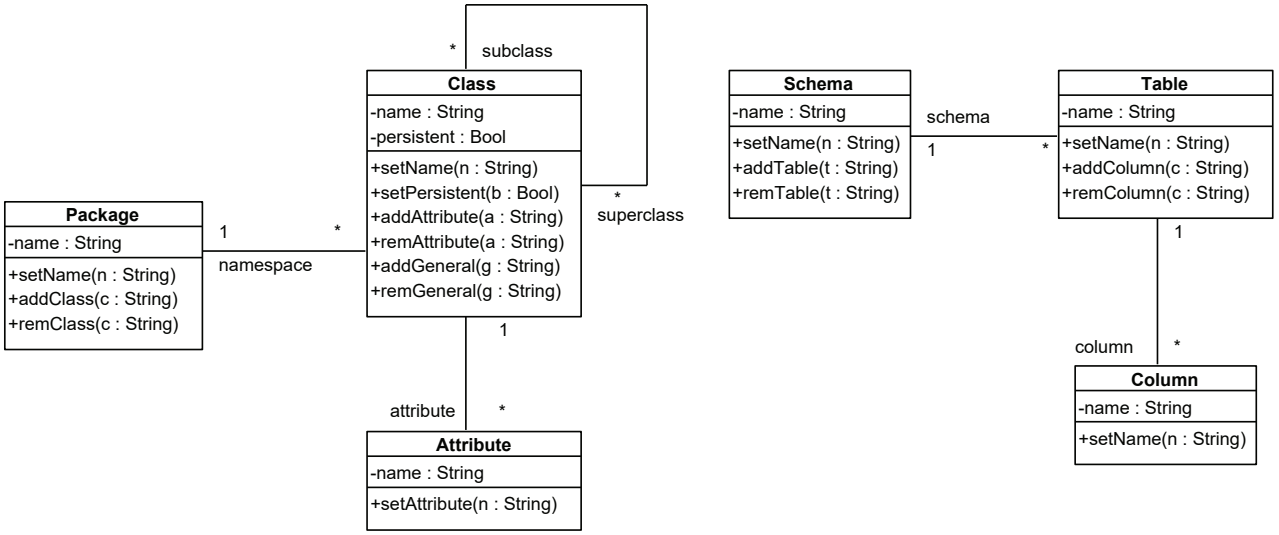

Figure 2. Class Diagrams of the Unified Modeling Language (UML) and Relational Database Management System (RDBMS) Metamodels (Adapted From [12]).

An illustrative example Here, a simplified version of the classic object-relational QVTr model transformation is presented to motivate the need for a formal model to resolve some existing issues on the presented QVTr specification [6], and to explain the application of the semantics. Adapted from [6, 12], Figure 2 depicts the two corresponding simplified metamodels of UML and RDBMS. Based on these metamodels, Listing 1 displays a simplified QVTr transformation of a UML to RDBMS model [6].

The transformation *UML2RDBMS* aims to map a persistent class of a given package to a table of a schema, which has the same name as the package. Corresponding to each attribute of a class including the inherited ones in a UML model, there should be a column in the peer table in the RDBMS model. Moreover, regarding the generalization association of the UML metamodel, the acyclic constraint must hold. In other words, each class of a given UML model cannot be simultaneously superclass and subclass in a generalization relation. This constraint cannot be captured neither from the displayed class diagram of Figure 2 nor the Listing 1. The two top relations *PackageToSchema* and *ClassToTable* map each package to a schema with the same name and each class to a table as well. Each class name space, i.e., its container package, and the corresponding table schema are preserved through calling *PackageToSchema* relation from the *when* clause of the *ClassToTable* relation with the concrete domain variables, here p and s. Calling the *AttributeToColumn* relation with the concrete domain variables of c and t in the *where* clause of the relation *ClassToTable* ensures that the attributes of class c are mapped to columns of table t. Additionally, by calling directly the *PrimitiveAttributeToColumn* and *SuperAttributeToColumn* relations from the *where* clause of the *AttributeToColumn* relation, the primitive attributes as well as the inherited ones from superclasses of the class c are mapped to the corresponding columns in the table t.

As stated by Macedo and Cunha [12], even the simplified version of this transformation specification has inherent ambiguities, e.g., the resolution of recursions on handling the class generalization hierarchies.

2.3 The QVT Standard Related Languages and Tools

This section presents a brief survey on the existing languages/tools implemented based on the QVT standard to motivate the need for a formalism of model transformation concepts as a foundation to implement tools and their continuous support. There exist several languages/tools in the model transformation field,

which have implemented the QVT language standard [26, 27] full or in part. To summarize, a categorization of the common characteristics of model transformations is presented in the following categories, C1–C8:

- C1, **Execution Direction**: Single/Unidirectional (S) or Double/Bidirectional (D); the model transformation is executed in one direction or it can be reversed too.
- C2, **Traceability**: Explicit/Manual (E), Implicit/Automatic (I), no support (N); the trace elements which determine the candidate model elements of source and target of the transformation are to be created explicitly by the user or implicitly by the tool.
- C3, **Development Type**: Prototype (P) or Full (F); the tool has been released as a prototype or it is a full version release.
- C4, **Transformation Type**: Model-to-Model (M2M), Model-to-Code (M2C), or Both; the source and target of the transformation are models (M2M) or the generated target model is source code (M2C). It must be mentioned that the source code per se is considered generally the lowest level model in MDD.
- C5, **The metamodel type support**: Endogenous (En), Exogenous (Ex), or Both (B); the source and target models of the transformation conform to the same metamodels (Endogenous) or different metamodels (Exogenous).
- C6, **Transformation approach**:
  Relational/Declarative (R). Declaring/relating the candidate model elements of the source and target models
  Operational/Imperative (O). Specifying the transformation execution as a sequence of actions/rules.
  Graph-based (G). Representing a transformation as a set of graph transformation rules which by applying the rules produces the output/target graph from the input/source one
  Hybrid (H). This approach combines declarative and operational approaches to maximize the advantages of the approaches and minimize the disadvantages of them
- C7, **Cardinality of the source and target models**: number of participating models in the source/target of the transformation which can be a single model or multiple models. In other words, whether the transformations are 1-to-1 (I), 1-to-N (II), N-to-1 (III), or N-to-N (IV). The category 'IV' covers other three subcategories too.
- C8, **Current status of the tool**: whether the tool has support and continued (C) or discontinued (D).

Table 1 summarizes the main characteristics of the QVT-based languages/tools. See Appendix 6.2 for a detailed description of these languages/tools.

Table 1. Characteristics of Model Transformation Languages. C1–C8 and the Table Content Decodings/descriptions Are Explanied in the Surrounding Text.

Language/Tool	C1	C2	C3	C4	C5	C6	C7	C8
mediniQVT [7]	D	E	P	M2M	B	R	IV	D
SmartQVT [10]	S	E	P	B	B	O	IV	D
QVTo-Eclipse [9]	S	E	P	B	B	O	IV	C
QVTr-XSLT [8]	S	I	P	B	B	R	IV	D
ModelMorf [28]	D	E	P	B	B	R	IV	D
Together [29]	S	I	P	M2M	B	G	IV	C
JQVT [30]	S	N	P	M2M	Ex	O	I,II	D
UMLX [31]	S	I	P	M2M	B	G	IV	C
UML-RSDS [25]	D	E	P	M2M	B	R	IV	C

As stated in [13] and [12], the incomplete and in some situations, the ambiguous semantics of the QVT standard [6] have slowed down the emergence of effective tool support for the QVTr language. In particular, the QVTr semantics in Annex B [6] is omissive with regard to the semantics of relation *when* and *where* clauses, and does not distinguish the cases of top relation and non-top relation execution. It appears only to cover the case of top relation execution with no relation calls in either the *when* or *where* clauses. On the other hand, the Relations to Core translation of the QVT standard [6] gives a detailed operational semantics of QVTr by a translation to the Core language. We will aim to reconcile these two alternative semantics in our own QVTr semantic model, using Z to provide a clearer presentation of the concepts, compared to the complex Relations to Core mapping.

2.4 The Z Specification Language: A Brief Introduction

The Z specification language [21] is based on the *Zermelo Frankel* set theory and mathematical logic. It has been used successfully for the design and specification of many projects in the last two decades [32–34]. Each system is specified by schemas which, can be either state schema (the structure of a system) or operation schema (the state changes or behavior of a system). The Z schema construct has two compartments: declarations and predicates. *Declaration* section is used to introduce variables/components and the *predicate* section involves predicates to enforce the desirable conditions, restrict the declarations/relations of the declared components to name a few [21].

To refactor and modularize the specification and design structure, it is possible to split complex schemas and integrate them by inclusion.

2.5 Why Z?

Some of the main reasons which motivate using this language of specification are as follows:

- Using the Z language constructs, specifically *schemas*, i.e., *state* and *operation*, it is straightforward to specify a formal model of a real system with a required abstraction level [35–37].
- Definition of new types in numerous ways such as *given sets*, *free types*, *axioms* to name a few, makes Z notation as one of the best choices for a software designer to specify an abstraction of a real system.
- Using quantification, $\forall / \exists$, on the predicate definition facilitates the specification of iterations over the defined collections, such as Sets, Bags, and Sequences of each model.
- In addition to maturity, expressiveness, providing good facilities for proof, the Z language is an abstract language. Hence, supporting tools can be developed more easily and characterizing the expected model transformation concepts will be performed in a direct manner without introducing any further detail [38].

The existing formal semantics literature [13, 15, 19] regarding the QVTr specification and its execution modes will be discussed in more detail in the related works (Section 4), after the presentation of the proposed formal model of this paper. In the following, the formal model of the main model transformation concepts is presented using the Z notation [21].

3 Formal Specification of the Model Transformation Concepts

Despite the various existing model transformation standards, the lack of formalism for the model transformation concepts makes it difficult to automate the construction of model transformation tools [14, 39]. Moreover, formal specification of the model transformations and the underlying concepts will have a great impact on the quality of the captured language/tool models in MDD [40, 41]. In this research, the Z notation is used to formalize the main concepts of model transformation languages.

As stated by Stevens [19], one of the main drawbacks for the QVT-based tool developers is the difficulties of understanding the existing QVT standard semantics [6]. In this paper we aim to resolve this issue by defining an explicit logical semantics for QVTr. To ensure the correctness of the specification and compliance with the expected requirements, the resulting model can be verified and validated by the existing tools, such as Z/EVES [42] and Alloy [43, 44]. In this research, the Z/EVES tool version 2.1 has been utilized for type-checking of the Z specifications and the Alloy Analyzer version 4.2 for validation of the presented formalism [45, 46].

Because the formal model of the presented model transformation concepts is based on the OMG MOF standard, a brief description of this metamodel is presented in Appendix 6.1.

3.1 Formal Model of Model Transformation Concepts in the Z Notation

Figure 3 depicts an abstract view of our formal model that is presented gradually in this section. A brief introduction to the Z notations which have been used in our presented formalism is presented in Appendix 6.4.

Given sets The following given sets or basic types are used in the specification of the following model transformation concepts.

$$[Name, Type, Value, Constraint]$$

A brief description of these types are as follows:

Name used to uniquely name the concepts such as model, package, class, domain, etc.

Type provides the data types, e.g, the primitive types such as Integer, String, etc.

Value refers to any value, either values of standard OCL primitive types such as Integer, Real, Boolean, String, or object values, instances of classes/metaclasses. Object instances x have values $x.f$ for each property (feature) f of the class of x. The value is of the type specified for the property.

Constraint refers to any restriction on the model elements which will be evaluated as a boolean value. The constraint also can be a relation invocation in the context of the *when* and *where* clauses of a given transformation relation.

Abbreviations *Variable* declares the variable sets, e.g., the attributes of a given class on a model, the variable set of a given transformation relation to name a few. It can be defined as $Name \times Type$, as the set of pairs of names and types by the following abbreviation:

$$Variable == Name \times Type$$

Free types The following free types are defined which are used in the specification of concepts. For a logical type, the *Boolean* free type is defined. To provide an appropriate message for the user, the *Report* free type is introduced. Finally, to model the two

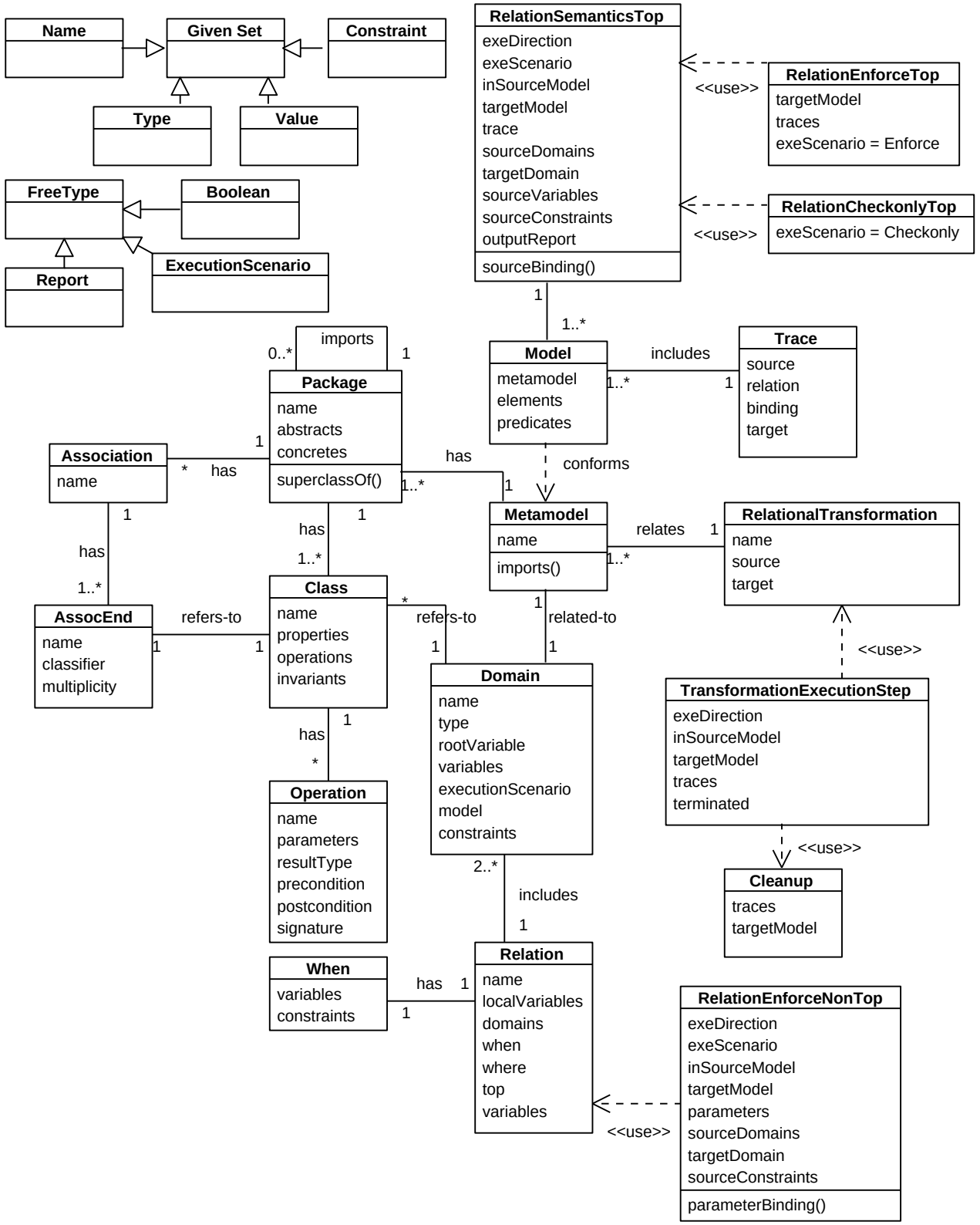

Figure 3. An Abstract View of the Presented Formal Model and Its Constituent Elements.

execution modes of a given model transformation, the *execScenario* free type will be used.

$$
\begin{aligned}
&Boolean ::= True \mid False\\
&Report ::= Check\_is\_Passed \mid Check\_is\_Failed \mid\\
&\qquad CREATE\_REPORT \mid\\
&\qquad DELETE\_REPORT\\
&execScenario ::= Checkonly \mid Enforce
\end{aligned}
$$

NamedElement This schema defines a valid named element:

$$
\begin{array}{|l}
\hline
NamedElement\\
name : Name\\
\hline
\end{array}
$$

Operation An operation has a name, signature, and pre and post condition:

$$
\begin{array}{|l}
\hline
Operation\\
NamedElement\\
params : \mathrm{seq}(Name \times Type)\\
resultType : Type\\
precondition : Constraint\\
postcondition : Constraint\\
signature : \mathrm{seq}(Type)\\
\hline
signature = \{i : \mathrm{dom}(params) \bullet\\
\qquad i \mapsto second(params(i))\}\\
\hline
\end{array}
$$

The signature is the sequence of parameter types which uniquely distinguishes methods from each other on their usage.

A *Class* includes a name, distinct attribute set, operations, and some constraints:

$$
\begin{array}{|l}
\hline
Class\\
NamedElement\\
properties : \mathbb{F}(Name \times Type)\\
operations : \mathbb{F}\ Operation\\
invariants : \mathbb{F}\ Constraint\\
\hline
\forall\, p_1, p_2 : Name \times Type \bullet\\
\quad \{p_1, p_2\} \subseteq properties \land p_1 \neq p_2 \Rightarrow\\
\qquad first\ p_1 \neq first\ p_2\\
\forall\, o_1, o_2 : Operation \bullet\\
\quad \{o_1, o_2\} \subseteq operations \land\\
\quad o_1.name = o_2.name \land\\
\qquad o_1.signature = o_2.signature \Rightarrow\\
\qquad\quad o_1 = o_2\\
\hline
\end{array}
$$

Association Each association which relates multiple classifiers is specified by the *Association* schema. The association relates a finite set of association ends.

$$
\begin{array}{|l}
\hline
Association\\
NamedElement\\
ends : \mathbb{F}_1\ AssocEnd\\
\hline
\#ends \geq 2\\
\hline
\end{array}
$$

The association ends per se are specified by the *AssocEnd* schema which involves a name, the connected classifier, and its multiplicity.

$$
\begin{array}{|l}
\hline
AssocEnd\\
NamedElement\\
classifier : Class\\
multiplicity : \mathbb{P}\,\mathbb{N}\\
\hline
\end{array}
$$

Package Each package which plays the role of a container to categorize and modularize the metamodel elements is specified by the *Package* schema. The specification includes a name, involved classes, abstract classes, concrete classes, associations, and generalization relationships defined by the *superclassOf* function. The predicate part of the schema enforces the uniqueness of the involved classes, associations; restriction of the associations as well as the abstract and concrete classes to the package defined classes. Moreover, in case of the *superclassOf* function, (1) the general class of a defined concrete class can be abstract/concrete, (2) the generalization relation is transitive, and also (3) it should not be reflexive.

$$
\begin{array}{|l}
\textit{Package} \\
\hline
\textit{NamedElement} \\
\textit{classes} : \mathbb{F}_1\ \textit{Class} \\
\textit{abstracts}, \textit{concretes} : \mathbb{F}\ \textit{Class} \\
\textit{assocs} : \mathbb{F}\ \textit{Association} \\
\textit{superclassOf} : \textit{Class} \nrightarrow \textit{Class} \\
\hline
\forall c_1, c_2 : \textit{Class} \mid \{c_1, c_2\} \subseteq \textit{classes} \bullet \\
\quad c_1.\textit{name} = c_2.\textit{name} \Rightarrow c_1 = c_2 \\
\forall a : \textit{Association};\ e : \textit{AssocEnd} \bullet \\
\quad a \in \textit{assocs} \land e \in a.\textit{ends} \\
\qquad \Rightarrow e.\textit{classifier} \in \textit{classes} \\
\forall a_1, a_2 : \textit{Association} \mid \\
\quad \{a_1, a_2\} \subseteq \textit{assocs} \bullet \\
\qquad a_1.\textit{name} = a_2.\textit{name} \Rightarrow a_1 = a_2 \\
\textit{abstracts} \subseteq \textit{classes} \\
\textit{concretes} \subseteq \textit{classes} \\
\textit{abstracts} \cap \textit{concretes} = \emptyset \\
\textit{abstracts} \cup \textit{concretes} = \textit{classes} \\
\mathrm{dom}\ \textit{superclassOf} \subseteq \textit{classes} \\
\mathrm{ran}\ \textit{superclassOf} \subseteq \textit{classes} \\
\forall c_1, c_2, c_3 : \textit{Class} \mid \{c_1, c_2, c_3\} \subseteq \textit{classes} \bullet \\
\quad \textit{superclassOf}(c_3) = c_2 \land \\
\qquad \textit{superclassOf}(c_2) = c_1 \Rightarrow \\
\qquad\quad \textit{superclassOf}(c_3) = c_1 \\
\forall c_1, c_2 : \textit{Class} \mid \{c_1, c_2\} \subseteq \textit{classes} \bullet \\
\quad \textit{superclassOf}(c_1) = c_2 \Rightarrow \\
\qquad \neg\ \textit{superclassOf}(c_2) = c_1
\end{array}
$$

Metamodel As stated earlier, each metamodel is a named model which involves some packages. Moreover, a metamodel can import other packages as well. The *import* relation is nonreflexive, i.e., a given package cannot import itself.

$$
\begin{array}{|l}
\textit{Metamodel} \\
\hline
\textit{NamedElement};\ \textit{pkgs} : \mathbb{F}_1\ \textit{Package} \\
\textit{imports} : \textit{Package} \leftrightarrow \textit{Package} \\
\hline
\mathrm{dom}\ \textit{imports} \subseteq \textit{pkgs} \\
\mathrm{ran}\ \textit{imports} \subseteq \textit{pkgs} \\
\forall p : \textit{Package} \bullet p \in \textit{pkgs} \Rightarrow \\
\qquad p \notin \mathrm{ran}(\{p\} \lhd \textit{imports})
\end{array}
$$

A model consists of a set of elements, and a set of predicates relating the elements. The model should conform to a specific metamodel.

$$
\begin{array}{|l}
\textit{Model} \\
\hline
\textit{metamodel} : \textit{Metamodel} \\
\textit{elements} : \mathbb{F}(\textit{Value}) \\
\textit{predicates} : \mathbb{F}(\textit{Constraint}) \\
\hline
(\textit{elements}, \textit{predicates}) \models \textit{metamodel}
\end{array}
$$

The conformance of each model to a metamodel is specified by the following axiomatic definition:

$$
\begin{array}{|l}
(\_, \_) \models \_ : (\mathbb{F}(\textit{Value}) \\
\times \mathbb{F}(\textit{Constraint})) \leftrightarrow \textit{Metamodel} \\
\hline
\forall\ \textit{mm} : \textit{Metamodel}; \\
\textit{elems} : \mathbb{F}(\textit{Value}); \\
\textit{preds} : \mathbb{F}(\textit{Constraint}) \bullet \\
\quad (\textit{elems}, \textit{preds}) \models \textit{mm} \Leftrightarrow \\
\qquad \forall\ e : \textit{elems} \Rightarrow \\
\qquad\quad \exists\ p : \textit{Package};\ c : \textit{Class} \mid \\
\qquad\quad p \in \textit{mm}.\textit{pkgs} \land \\
\qquad\quad c \in p.\textit{metaclasses} \bullet \\
\qquad\qquad (\textit{preds} \Rightarrow c.\textit{invariants}) \land \\
\qquad\qquad e \leadsto c \land \\
\qquad\qquad \textit{eval}(\textit{elems}, \textit{preds}) = \textit{True}
\end{array}
$$

$$
\begin{array}{|l}
\textit{eval} : \mathbb{F}(\textit{Value}) \times \mathbb{F}(\textit{Constraint}) \nrightarrow \textit{Boolean}
\end{array}
$$

A model m conforms to a given metamodel mm if and only if each model element has its metaelement defined in mm [47]. The statement $e \leadsto c$ means that the model element e is an instance of the class c from the package p of the metamodel mm. In addition, the invariants of c should be true for e.

The *elements* include both objects and primitive values (strings, numerics, booleans).

The *predicates/constraints* define the types and properties of the values. We assume these are of three forms $x : T$, $e1.f = e2$, $e1.f \rightarrow includes(e2)$. The predicate

$$x : T$$

declares the type of element x, where T is an OCL primitive type (*Integer*, *Real*, *String*, *Boolean*) or a concrete class from $metamodel.pkgs.classes$.

A predicate

$$x.f = val$$

for object-valued x asserts that feature f of element x has value val, where val is an element. For the conformance relation $\models$ to hold, f must be a feature of the class of x, and the equated value val must be of the correct type for this feature according to the metamodel. The predicate is true iff the value of the feature f of x is equal to the value of val.

Likewise, a predicate

$$x.f \rightarrow includes(val)$$

asserts that element val is a member of the collection-valued feature f of x. f must satisfy the typing and multiplicity restrictions given in the metamodel.

If features f and g are two opposite ends of the same bidirectional association r, then a predicate on f entails the presence of a corresponding predicate on g. For example, if r is many-many, $e2.g \rightarrow includes(e1)$

is in the predicates of a model iff $e1.f{\rightarrow}includes(e2)$ is in the predicates.

A function

$$elems : Constraint \rightarrow \mathbb{F}(Value)$$

gives the set of elements referred to in a constraint.

For example, a conforming model for a metamodel containing a single class A with an integer attribute att could have

$$elements = \{a, b, 5, -3\}$$
$$predicates = \{a : A, b : A, 5 : Integer, -3 : Integer, a.att = 5, b.att = -3\}$$

3.2 Formal Specification of QVTr Syntax

The relations of a QVTr transformation are defined by some optional pre/post conditions specified through "when"/"where" constraints respectively as well as a two-way relation (in case of a bidirectional transformation) between domains from source and target, respectively. Primitive domains which are neither "checkonly" nor "enforce" are used to pass some configuration information or constants to the relation. Each transformation can have two kinds of relation: top or non-top. The successful completed execution of a transformation requires that all the top level relations hold but the non-top level relations only need to hold for specific parameter values when they have been invoked with these values directly or transitively from the *where clause* of another relation. Moreover, a relation can define some local variables which are used as variables of other parts such as domain patterns and when/where clauses within the relation.

Each relation may be executed in "checkonly" or "enforce" execution mode, and with a specific model as its execution target (direction). In the "checkonly" execution mode, the relation is checked to see whether it can be established in the execution direction. But for enforce execution in the direction of a model of target domains marked "enforce" then the related model elements are created, modified, or deleted to enforce a consistent target model according to the domain and relation constraints. Each domain is applied and matched against a given specific model type (i.e., a metamodel). The domain model type is represented by the *type* variable in the *Domain* schema below.

The domains of a relation reference elements of the models involved in the relation. For example, the checkonly domain

```
checkonly domain uml c:Class {
    persistent = true,
    namespace = p: Package{},
    name = n
}
```

is defined in relation ClassToTable in the UML2RDBMS transformation. This refers to objects c and p of classes *Class* and *Package* in the *UML* metamodel, and values $n : String$ and $true : Boolean$.

The domain concept is formalised as:

Domain

$NamedElement$
$type : Class$
$rootVar : Variable$
$vars : \mathbb{F}(Variable)$
$exeScenario : execScenario$
$model : Metamodel$
$constraints : \mathbb{F}(Constraint)$

$rootVar = (name, type)$
$rootVar \in vars$
$\exists\, p : Package \bullet p \in model.pkgs \wedge$
$\quad type \in p.classes$

The *vars* are all the variables explicitly or implicitly declared in the domain, including the domain root variable and variables of object template expressions contained in the domain pattern. The *constraints* include the explicit predicates in the domain pattern and condition, and typing predicates for the domain variables. For example, the above domain has constraints

$$c : Class, c.persistent = true,$$
$$p : Package, c.namespace = p, c.name = n$$

We assume that the constraints can be expressed in the three forms identified above for models. Some other forms of constraint, such as $s.f{\rightarrow}includesAll(t)$, can be reduced to the above forms when evaluated on specific models.

The *when* clause of a relation defines restrictions over application of the relation, including (for top relations) checks that another relation has been previously established for specific elements. Apart from the constraint forms $v : T$, $v.f = w$, $v.f{\rightarrow}includes(w)$, we also permit negations of these constraint forms in the *when* clause, and tests $v : T$ where T is abstract. The *when* clause predicates will never be enforced, only evaluated.

When

$vars : \mathbb{F}(Variable)$
$constraints : \mathbb{F}(Constraint)$

The *vars* are all variables used in the *when* clause, in particular including the parameters of relation calls $R(x, y)$. They correspond to *when_variable_set* in Annex B of the QVTr v1.3 standard.

A relation has a sequence of at least two domains.

The order of domains is significant in matching calls of the relation to its definition.

Relation

$$
\begin{array}{l}
NamedElement \\
localVars : \mathbb{F}(Variable) \\
domains : \text{seq}(Domain) \\
when : When \\
where : \mathbb{F}(Constraint) \\
top : Boolean \\
vars : \mathbb{F}(Variable) \\
\hline
vars = \bigcup\{d : Domain \mid \\
\quad d \in \text{ran } domains \bullet d.vars\} \cup \\
\quad\quad localVars \cup \\
\quad\quad\quad when.vars \\
\#domains \geq 2
\end{array}
$$

The *vars* correspond to *R_variable_set* in Annex B of the QVTr v1.3 standard.

The *where* clause of a relation is treated as a set of predicates which specify updates to the target model. These may be assignments $v.f = w$ or additions $v.f \rightarrow includes(w)$ to features f, for variables w (source or target) and v (target). Calls to enforce non-top relations $R(p1, ..., pn)$ may also occur in the *where* clause.

3.3 Formal Specification of the QVTr Semantics

A key concept in the semantics is the concept of a *binding*, which is a partial function from *Variable*s to *Value*s:

$$Binding == Variable \nrightarrow Value$$

Bindings $g : Binding$ link the syntax of a transformation specification to elements in the models that it operates on. Typically, $\text{dom}(g)$ is a subset of the variables in a transformation rule r (such as a QVTr relation) and $\text{ran}(g) \subseteq m.elements$ for some model m associated with r. If $g((name, type)) = elem$, then the type of *elem* in m must be consistent with *type*, i.e., variables designated as integers must map to integer elements, etc. In addition, if *type* is a class in metamodel *mm*, then *elem* must be of the same type in m, where $m.metamodel = mm$.

Each application of a rule r will involve a binding f of the source/input variables of r to elements of the source model, which satisfies the application conditions of r, and the extension of f to a binding g of all r's variables to the source and target models, such that the constraints of r hold true for the models wrt g. To satisfy the application conditions, source bindings may also need to be made to target elements referenced in relation calls in when clauses.

A relation call $R(p1, ..., pn)$ in a *when* clause is logically interpreted as a test for an occurrence of R in a trace sequence, and the bindings of the domain root variables $d1, ..., dn$ in any such occurrence are then used for $p1$, ..., pn respectively. Traces are essentially a sequence of tuples (s, r, b, t) where s is the source model, r the relation applied in the step, and b the binding used to apply r to produce the target model t.

Trace

$$
\begin{array}{l}
source : Model \\
relation : Relation \\
binding : Binding \\
target : Model \\
\hline
\text{ran}(binding) \subseteq \\
\quad source.elements \cup target.elements
\end{array}
$$

The *csetEval* function is used to evaluate the satisfaction of a given set of constraints in a trace sequence and a model, wrt a binding.

$$
\begin{array}{l}
csetEval : \mathbb{F}\ Constraint \times Binding \times \\
\quad\quad \text{seq } Trace \times Model \rightarrow Boolean \\
\hline
csetEval = (\lambda\, s : \mathbb{F}\ Constraint; \\
\quad f : Binding;\ ts : \text{seq } Trace;\ m : Model \bullet \\
\quad\quad \forall\, c : Constraint \mid c \in s \bullet \\
\quad\quad\quad eval(c, f, ts, m) = True)
\end{array}
$$

$$
\begin{array}{l}
eval : Constraint \times Binding \times \\
\quad\quad \text{seq } Trace \times Model \rightarrow Boolean
\end{array}
$$

$csetEval(cs, f, tr, m)$ evaluates to *True* if each constraint $c \in cs$ is satisfied in m and tr wrt f.

The following schema defines the common aspects between different execution modes (checkonly or enforce) of a top relation:

RelationSemanticsTop

$\Xi Relation$
$direction? : Model$
$exeScenario : execScenario$
$m_s?, m_t : Model$
$tr : \text{seq}(Trace)$
$srcDoms : \mathbb{F}(Domain)$
$tDom : Domain$
$srcVars : \mathbb{F}(Variable)$
$scrConstraints : \mathbb{F}(Constraint)$
$srcBinding : Variable \nrightarrow Value$
$Rep! : \mathbb{F}(Report)$

$top = True$
$srcDoms = \{d : Domain \mid$
$\quad d \in \text{ran } domains \bullet$
$\quad\quad d.model \neq direction?\}$
$tDom \in \text{ran } domains$
$tDom.model = direction?$
$srcVars = \bigcup\{d : srcDoms \bullet$
$\quad\quad d.vars\} \cup$
$\quad\quad localVars \cup$
$\quad\quad when.vars$
$scrConstraints = \bigcup\{d : srcDoms \bullet$
$\quad\quad d.constraints\} \cup$
$\quad\quad when.constraints$
$\#tr > 0 \Rightarrow m_t = last(tr).target$
$\#tr = 0 \Rightarrow m_t = \emptyset$
$srcBinding \in srcVars \rightarrow$
$\quad\quad m_s?.elements \cup m_t.elements$
$csetEval(scrConstraints,$
$\quad\quad srcBinding, tr, m_s?) = True$

direction? is the designated model to be updated, i.e.., the execution direction of the relation. $m_s?$ is the source input model at start of execution/checking of the relation. m_t is the target input model at the start of the relation. It is taken to be the final target model produced by preceding rule applications, if there are any, and otherwise to be the empty model $\emptyset$. *tr* is the existing trace history of preceding relation executions. *srcDoms* are the domains whose model is not *direction*?, whilst *tDom* is the domain with model *direction*?. *srcVars* are all variables to be instantiated from the elements of the input models, including all variables of source domains, local variables and the variables of the *when* clause. *scrConstraints* are the constraints of the source domains and *when* clause, which the source variables should satisfy when bound by *srcBinding* to elements of the input models. $\emptyset$ denotes the empty model with empty elements and predicate sets.

The following schema defines the possible form of an enforce execution of a single top relation, in the direction of a particular model. It covers the case where new elements may be created and the input target model m_t is extended with these to form the output target model m_t'.

RelationEnforceTop

$RelationSemanticsTop$
$m_t' : Model$
$tr' : \text{seq}(Trace)$

$exeScenario = Enforce$
$tDom.exeScenario = Enforce$
$\exists\, m : Model \bullet m.metamodel = direction? \wedge$
$\quad \exists\, g : vars \rightarrow (m_s?.elements \cup$
$\quad\quad\quad elements) \bullet$
$\quad\quad srcVars \lhd g =$
$\quad\quad\quad srcVars \lhd srcBinding \wedge$
$\quad\quad csetEval(tDom.constraints \cup$
$\quad\quad\quad where, g, tr, m) = True \wedge$
$\quad\quad m_t' = m \wedge$
$\quad\quad CREATE\_REPORT \in Rep! \wedge$
$\quad\quad \exists\, t : Trace \bullet t.source = m_s? \wedge$
$\quad\quad\quad t.relation.name = name \wedge$
$\quad\quad\quad t.binding = g \wedge$
$\quad\quad\quad t.target = m \wedge$
$\quad\quad\quad tr' = tr \frown \langle t \rangle$

A constraint implied by the QVTr standard is that the *srcBinding* should not have occurred previously for this relation in the trace, i.e.

$$\neg\, (\exists\, t : \text{ran}(tr) \bullet t.relation.name = name \wedge srcBinding \subseteq t.binding)$$

This means that relations should not be re-applied to arguments for which they have already been established.

An occurrence of a relation test $R(p1, ..., pn)$ for a top-level relation in the *when* clause is checked against *tr* by $csetEval(scrConstraints, srcBinding, tr, m_s?)$, which includes the condition:

$$\begin{aligned}&\exists\, t : \text{ran}(tr) \bullet \\ &\quad t.relation.name = R.name \\ &\quad srcBinding(p1) = t.binding(d1) \\ &\quad\quad \vdots \\ &\quad srcBinding(pn) = t.binding(dn)\end{aligned}$$

Where R has domain sequence $dm_1, ..., dm_n$ with root variables $d1, ..., dn$.

Our semantics does not prescribe how m in *RelationEnforceTop* should be chosen. One principle, of least change, suggests that m should be minimal such that a suitable g can be found, and of minimal difference to m_t [1]. A concept of ordering and

[1] Annex B of the QVT standard [6] is ambiguous regarding the operational interpretation of element creation via target

difference of models can be based on the subset and subtraction relations of the element and predicate sets. Thus new elements should be only introduced when there are no existing elements in m_t which satisfy a required predicate. This means that g must use existing m_t elements where possible.

In addition, m_t' should preserve the preceding target bindings, i.e., a relation application cannot remove elements which have been bound by preceding relation applications:

$$t \in \text{ran}(tr) \Rightarrow \forall\, x : t.target.elements \bullet x \in \text{ran}(t.binding) \Rightarrow x \in m_t'.elements$$

Predicates established by preceding applications should also be preserved:

$$t \in \text{ran}(tr) \Rightarrow \forall\, c : t.target.predicates \bullet elems(c) \subseteq \text{ran}(t.binding) \Rightarrow c \in m_t'.predicates$$

This means that element features should not be re-assigned different values by different relation applications, however, new elements can be added to collection-valued features by different relation applications.

We show that there is a solution to the above restrictions, by defining a process for constructing m and g for cases of relations without *where* clauses as follows:

(1) Initialise m to m_t and g to *srcBinding*
(2) Each target domain constraint $c \in tDom.constraints$ is either of form $v.f = w$ for variables v and w and feature f, or of form $v : T$ or $v.f \rightarrow includes(w)$.
(3) In the case of $v : T$ for type T:
  (a) If there is no element e of m with constraint $e : T$ in $m.predicates$, add a new e to $m.elements$, with the constraint $e : T$ and extend g by the binding $v \mapsto e$.
  (b) If there is an existing e in $m.elements$ which satisfies all of the required constraints of v, extend g by the binding $v \mapsto e$.
  (c) Otherwise, create a new e and proceed as in case (a).
(4) In the case of $v.f = w$, if there are existing elements $e1$, $e2$ with $g(v) = e1$, $g(w) = e2$:
  (a) If $e1.f = e2 \in m.predicates$ then no updates are needed
  (b) If $e1.f = e3 \in m.predicates$ for $e3 \neq e2$, there is a conflict between relation applications and no change should be made
  (c) If there is no predicate $e1.f = e \in m.predicates$ for any e, then extend $m.predicates$ by the predicate $e1.f = e2$.

  Otherwise, add new $e1$ and/or $e2$ to m and bindings $v \mapsto e1$, $w \mapsto e2$ to g, and predicate $e1.f = e2$ to $m.predicates$, together with type constraints for $e1$ and/or $e2$. In each case, a predicate on the opposite end feature of f may also need to be added, if f is one association end of a bidirectional association.
(5) The case of $v.f \rightarrow includes(w)$ is similar, with a constraint $e2 : e1.f$ being added for existing or new elements.

object templates. It asserts that target elements identified by a key value are looked-up by that value and re-used if they already exist, or created if they do not exist. However, regarding objects without key properties it leaves open different possibilities, including: (i) always creating new objects, or (ii) selecting existing objects where possible.

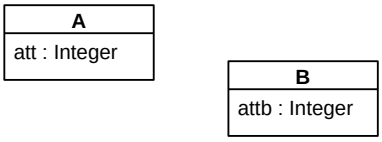

Figure 4. Example Metamodels.

As an example, consider the case of a metamodel $MM1$ with a single class A, and metamodel $MM2$ with single class B (Figure 4).

A possible top relation with source model an instance *src* of $MM1$ and target model an instance *trg* of $MM2$ could be:

```
top relation R {
   checkonly domain src ax : A { att = attvalue };
    enforce domain trg bx : B { attb = attvalue };
}
```

Consider the case of executing a transformation which consists of this single rule, on an initial $MM1$ model with elements $\{a1, a2, 1\}$, and constraints $\{a1 : A, a2 : A, 1 : Integer, a1.att = 1, a2.att = 1\}$. This is m_s? for the first application of R, and the *srcBinding* for this application could be:

$$\{ax \mapsto a1, attvalue \mapsto 1\}$$

The initial target model m is the empty model $\emptyset$. The initial value of g is *srcBinding*. To complete the construction of g and m in the definition of *RelationEnforceTop*, we consider the target domain constraints $bx : B$ and $bx.attb = attvalue$. Since m is initially empty, by clause 3(a), a new element $b1$ is added to $m.elements$, constraint $b1 : B$ added to $m.predicates$, and binding $bx \mapsto b1$ added to g.

For constraint $bx.attb = attvalue$, clause 4(c) applies, and $m.predicates$ is extended with a new constraint $b1.attb = 1$. Thus the resulting target model m_t' has

$$elements = \{b1, 1\}$$
$$predicates = \{b1 : B, b1.attb = 1\}$$

Now, R is still enabled for execution on the $a2$ element (but not on $a1$ because the source bindings for $a1$ would already occur in the trace). For this application the source binding is

$$\{ax \mapsto a2, attvalue \mapsto 1\}$$

The target model m starts as the preceding m'_t. Now clause 3(b) applies, because $b1$ already satisfies the required constraints for bx, so the target model is not changed, and g can be extended with the binding $bx \mapsto b1$.

Execution of the *where* clause of a relation can lead to a series of intermediate models m' between m_t and m in which m is progressively constructed (no new variables should be introduced in the where clause, so g is not further extended). The *where* predicates are not specifically ordered, however some execution order should be chosen so that where possible a valid execution results. If the predicates are ordered as w_1; ...; w_r then intermediate target models m'_j are created where m'_1 is the m produced by considering the target domain constraints as described above, and m'_{j+1} is produced from m'_j by satisfying constraint w_j.

More precisely, a *where* predicate

$$v.feature = w$$

for variables v, w updates m with the constraint $velem.feature = welem$, where $velem$ is the current bound value of v, and $welem$ of w. Similarly for $v.feature \rightarrow includes(w)$ and $v.feature \rightarrow includesAll(w)$. A call $R(p1, ..., pn)$ of a non-top relation R with domain root variables $d1$, ..., dn carries out an enforce execution of R with $d1$ bound to the current bound value of $p1$, ..., dn to the bound value of pn. This execution takes the current state of m as its m_t input. It returns an updated m as its m'_t output. *RelationEnforceNonTop* is similar to *RelationEnforceTop* but makes no reference or update to traces. We assume there are no additional variables in the *when* clause, and no tests on the trace in that clause:

RelationEnforceNonTop

$\Xi Relation$
$direction? : Metamodel$
$exeScenario : execScenario$
$m_s? : Model$
$m_t, m'_t : Model$
$params : \text{seq}(Variable)$
$srcDoms : \mathbb{F}(Domain)$
$tDom : Domain$
$srcVars : \mathbb{F}(Variable)$
$scrConstraints : \mathbb{F}(Constraint)$
$paramBinding? : Variable \nrightarrow Value$

$top = False$
$srcDoms = \{d : Domain \mid d \in \text{ran } domains \wedge d.model \neq direction?\}$
$tDom \in \text{ran } domains$
$tDom.model = direction?$
$srcVars = \bigcup\{d : srcDoms \bullet d.vars\} \cup localVars \cup \text{ran}(params)$
$scrConstraints = \bigcup\{d : srcDoms \bullet d.constraints\} \cup when.constraints$
$\forall d : Domain \bullet d \in \text{ran } domains \Rightarrow d.rootVar \in \text{ran } params$
$\forall i : \text{dom } params \bullet params(i) = domains(i).rootVar$
$\text{dom}(paramBinding?) = \text{ran}(params)$
$\exists m : Model \bullet m.metamodel = direction? \wedge$
$\quad \forall f : srcVars \rightarrow (m_s?.elements \cup m_t.elements) \bullet$
$\quad\quad \text{ran}(params) \lhd f = paramBinding? \wedge$
$\quad\quad csetEval(scrConstraints, f, \langle\rangle, m_s?) = True \Rightarrow$
$\quad\quad\quad \exists g : vars \rightarrow (m_s?.elements \cup m.elements) \bullet$
$\quad\quad\quad\quad srcVars \lhd g = srcVars \lhd f \wedge$
$\quad\quad\quad\quad csetEval(tDom.constraints \cup where, g, \langle\rangle, m) = True \wedge$
$\quad m'_t = m$

This means that if non-top relation R is called with parameter values a, b, corresponding to its domain root variables s, t, a binding $s \mapsto a, t \mapsto b$ is used as the *paramBinding* for the call, and all extensions f of this binding to the other source variables of R, such that f satisfies the source constraints, must be extensible to a binding g of all variables of R, which satisfies all of R's constraints.

For check-only semantics, we simply test for the existence of a suitable target model:

RelationCheckonlyTop

RelationSemanticsTop

$$
\begin{aligned}
&exeScenario = Checkonly \\
&(\exists\, m : Model \bullet \\
&\quad m.metamodel = direction? \land \\
&\quad \exists\, g : vars \\
&\qquad \rightarrow (m_s?.elements \cup m.elements) \bullet \\
&\qquad srcVars \lhd g = \\
&\qquad\quad srcVars \lhd srcBinding \land \\
&\qquad csetEval(tDom.constraints \cup \\
&\qquad\quad \mathrm{ran}(where), g, tr, m) = \\
&\qquad\quad True) \Rightarrow \\
&\qquad Check\_is\_Passed \in Rep! \land \\
&\neg\, (\exists\, m : Model \bullet \\
&\quad m.metamodel = direction? \land \\
&\quad \exists\, g : vars \\
&\qquad \rightarrow (m_s?.elements \cup m.elements) \bullet \\
&\qquad srcVars \lhd g = \\
&\qquad\quad srcVars \lhd srcBinding \land \\
&\qquad csetEval(tDom.constraints \cup \\
&\qquad\quad \mathrm{ran}(where), g, tr, m) = \\
&\qquad\quad True) \Rightarrow \\
&\qquad Check\_is\_Failed \in Rep!
\end{aligned}
$$

According to the QVT-R standard, there is an implicit deletion of target elements which have not been matched to any source element via any relation. Therefore we define a 'cleanup' phase at the end of a transformation execution, which removes target model elements which do not appear in the range of any g binding in the trace targets:

Cleanup

$$
\begin{aligned}
&tr, tr' : \mathrm{seq}(Trace) \\
&m_t, m_t' : Model \\
&\#tr > 0 \Rightarrow \\
&\quad m_t = last(tr).target \land \\
&\quad (\textbf{let}\ unusedElements == \\
&\qquad \{x : m_t.elements \mid \forall\, t : \mathrm{ran}(tr) \bullet \\
&\qquad\quad x \notin \mathrm{ran}(t.binding)\} \bullet \\
&\qquad m_t'.elements = m_t.elements - \\
&\qquad\qquad unusedElements) \\
&\#tr = 0 \Rightarrow m_t' = m_t \\
&tr' = tr
\end{aligned}
$$

The model predicates of m_t which refer to any element in *unusedElements* are also removed, in the case of $\#tr > 0$.

A relational transformation with two models can be formalised as:

RelationalTransformation

$$
\begin{aligned}
&name : Name \\
&src, trg : Metamodel \\
&relations : \mathbb{F}(Relation) \\
&\forall\, r : relations \bullet \\
&\qquad \forall\, d : Domain \mid \\
&\qquad\quad d \in \mathrm{ran}\, r.domains \bullet \\
&\qquad\qquad d.model \in \{src, trg\}
\end{aligned}
$$

The execution of a transformation is then the composition of individual relation applications, while any application is enabled, and is terminated by *Cleanup*:

TransformationExecutionStep

$$
\begin{aligned}
&\Xi RelationalTransformation \\
&direction? : Model \\
&m_s? : Model \\
&m_t, m_t' : Model \\
&tr, tr' : \mathrm{seq}(Trace) \\
&terminated, terminated' : Boolean \\
&terminated = False \\
&direction? \in \{src, trg\} \\
&(\exists\, r : relations \bullet \\
&\quad r.RelationCheckonlyTop \land \\
&\qquad Check\_is\_Passed \in Rep! \Rightarrow \\
&\qquad\quad r.RelationEnforceTop \land \\
&\qquad\quad terminated' = False) \\
&(\neg\, (\exists\, r : relations \bullet \\
&\quad r.RelationCheckonlyTop \land \\
&\qquad Check\_is\_Passed \in Rep!) \Rightarrow \\
&\qquad\quad (Cleanup \land \\
&\qquad\qquad terminated' = True))
\end{aligned}
$$

3.4 Type Checking of the Formal Model

In this research, the Z/EVES 2.1 tool was used to type check the presented formal model of the transformation concepts. Numerous errors related to the used and declared given types, schema predicates specifically the universal quantifier applications are found and resolved. Then, to verify the correctness of the state schemas, the satisfaction of their *preconditions* was investigated. The *true* result ensures that the schema is valid in all of its conditions.

3.5 Validating the Presented Formalism Using the Alloy Analyzer Tool

Alloy is a language and tool which evaluates a model by restricting its instances and constructing their representing satisfiability formulas which can be solved by some integrated SAT solvers of this tool [48]. We use the Alloy Analyzer since the Alloy language is very similar to the Z notation. Both Alloy and Z are based on the first-order logic and set theory [44].

Unlike many theorem provers for Z, Alloy can be used to analyze the formal models of the Z notation without any full experience and knowledge regarding this tool analysis steps [45]. Alloy Analyzer is free and supported by a well research group from MIT and some online forums [2]. Moreover, the syntax conversion from the Z notation to input to the Alloy Analyzer is straightforward [45, 46]. The validated Alloy model of our formalism is presented in Appendix 6.3.

3.6 The Classic Object-Relational Model Transformation Specification

To show the applicability of the proposed formalism, the classic simplified object-relational model transformation is specified in the metamodel level. Referring to Figure 2 and recalling the *Metamodel* schema specification, first the metamodels will be instantiated as following [49].

UMLMetamodel ::= [name = 'UML',
  pkgs = {ClassDiag},
  imports = ∅]

ClassDiag ::= [classes = {Package, Class, Attribute},
  abstracts = ∅,
  concretes = {Package, Class, Attribute},
  assocs = { (Package,Class), (Class, Class),
    (Class, Attribute)},
  superclassOf = ∅]

RDBMSMetamodel ::= [name = 'RDBMS',
  pkgs = { DBSchema }, imports = ∅]

DBSchema ::= [classes = {Schema,
    Table, Column},
  abstracts = ∅,
  concretes = {Schema, Table, Column},
  assocs = {(Schema, Table), (Table, Column)},
  superclassOf = ∅]

Again, by referring to the *Relation* and *Domain* schemas, Figure 2, and Listing 1, the transformation including its constituent relations are instantiated as follows.

Relation ::= (name = PackageToSchema,
  localVars = {(n, String)},
  domains = [(name = p, vars = {(p, Package)},
    model = UMLMetamodel, type = Package,
    constraints = {p:Package, n : String,
      p.name = n}),
    (name = s, vars = {(s, Schema)},
    model = RDBMSMetamodel,
    type = Schema,
    constraints = {n: String, s : Schema,
      s.name = n})]
  when = (∅, ∅),
  where = ⟨⟩,
  top = true)

Relation ::= (name = ClassToTable,
  localVars = {(n, String)},
  domains = [(name = c,
    vars = {(c, Class), (p, Package)},
    model = UMLMetamodel,
    type = Class,
    constraints = {c.persistent = true,
      c.namespace = p, c.name = n}),
    (name = t, vars = {(t,Table), (s,Schema)},
      model = RDBMSMetamodel,
      type = Table,
      constraints = {t.schema = s, t.name = n})]
  when = ({(p, Package), (s, Schema)},
    [PackageToSchema(p, s)]),
  where = {AttributeToColumn(c, t)},
  top = true)

Relation ::= (name = AttributeToColumn,
  localVars = ∅,
  domains = [(name = c,
    vars = {(c, Class)},
    model = UMLMetamodel,
    type = Class,
    constraints = {c : Class}),
    (name = t, vars = {(t, Table)},
      model = RDBMSMetamodel,
      type = Table,
      constraints = { t : Table })]
  when = (∅, ⟨⟩),
  where = [PrimitiveAttributeToColumn(c, t),
    SuperAttributeToColumn(c, t)],
  top = false)

Relation ::= (name = PrimitiveAttributeToColumn,
  localVars = {(n, String)},
  domains = [(name = c,
    vars = {(c, Class), (a, Attribute)},
    model = UMLMetamodel,
    type = Class,
    constraints = {a ∈ c.attribute,
      a.name = n, c : Class, a : Attribute}),
    (name = t, vars = {(t, Table), (cl, Column)},
      model = RDBMSMetamodel,
      type = Table,
      constraints = {cl ∈ t.column,
        cl.name = n, cl : Column, t : Table})]

[2] http://alloytools.org/community.html

when = (∅, ⟨⟩),
where = ⟨⟩,
top = false)

Relation ::= (name = SuperAttributeToColumn,
localVars = { (n, String) },
domains = [(name = c,
vars = { (c,Class), (g,Class) },
model = UMLMetamodel,
type = Class,
constraints = {g ∈ c.superclass,
c : Class, g : Class}),
(name = t, vars = {(t, Table)},
model = RDBMSMetamodel,
type = Table,
constraints = {t : Table })]
when = (∅, ∅),
where = [AttributeToColumn(g, t)],
top = false)

For an example of the top-level enforce execution semantics, for *PackageToSchema*, $m_s?$ could have elements $\{`x', pk\}$ and predicates $\{`x' : String, pk : Package, pk.name = `x'\}$. The only possible *srcBinding* is $\{n \mapsto `x', p \mapsto pk\}$.

m_t' can have elements $\{`x', sc\}$ and predicates $\{`x' : String, sc : Schema, sc.name = `x'\}$, $g$ is $srcBinding \cup \{s \mapsto sc\}$. A new element $sc$ has been added, with the predicate $sc.name = `x'$, in order to validate the target constraint $s.name = n$.

For non-top execution, if class cx had attributes $a1$, $a2$ with names $n1$, $n2$, in source model $m_s?$ supplied to a call $PrimitiveAttributeToColumn(cx, tx)$ with tx being a table already created to correspond to cx, then the binding $paramBinding? = \{c \mapsto cx, t \mapsto tx\}$ of the call will be extended to

$$f1 = paramBinding? \cup \{a \mapsto a1, n \mapsto n1\}$$

and

$$f2 = paramBinding? \cup \{a \mapsto a2, n \mapsto n2\}$$

For each of $f1$, $f2$ there must be extensions $g1$, $g2$ which bind cl to new or existing *Column* objects with names $n1$, $n2$ and contained in tx. The resulting target model must contain all the new objects and satisfy all relation constraints for each $g1$, $g2$.

3.7 Resolving issues in the QVTr semantics

The semantics can be used to make precise issues in QVTr semantics and to propose resolutions for these.

For example, issue QVT14-55 "Check before enforce is unsound" (https://issues.omg.org/issues/spec/QVT/1.3) identifies that the QVTr check-before-enforce concept is impractical in general.

We address check-before-enforce in the process for constructing a new model m_t' for a rule execution in *RelationEnforceTop* and *RelationEnforceNonTop*: clause 3(b) states that if an existing target element $e \in m.elements$ satisfies all the required target domain constraints for target variable v, then the binding $v \mapsto e$ can be added to the overall relation binding g, and no new element needs to be introduced to m to satisfy the v constraints.

This is possible in simple cases where the constraints are equalities $v.f = w$ of v attributes to values w such as numbers and strings. We gave an example in Section 3.3. However, as issue QVT14-55 points out, if further elements $e1$, $e2$, etc are required to exist by constraints such as $v.r \rightarrow includes(w)$, the complexity of determining if such elements already exist can become impractical.

There are alternative semantics which could be considered for check-before-enforce, and could be defined using our formalism:

- *Key-based lookup*: If there is an existing element e with the same key attribute value $e.key = v$ as required by the target domain constraints for variable x, then x must be bound to e. Any conflicts in other feature values between established predicates for e and domain constraints of x indicate an inconsistent specification.
- *Check-before-enforce*: If there is an existing element e which already satisfies all the required constraints for x, then bind x to e.
- *Least-change update*: If there is an existing element e which already satisfies some (at least 1) required constraint for x, and has no conflicts with other required constraints, bind x to e and perform necessary updates of e's features to satisfy the required constraints.
- *Always create new target elements*: If there are no key features of the required target variable x, create a new element e irrespective of already-existing elements.

The least-change update always reuses existing elements wherever possible, whilst the final option only reuses elements in cases where this is required by key features. The first two options are included in the standard QVTr semantics, the third and fourth options could perhaps be notated by additional domain qualifiers.

Related to this issue, an important point that is implied but never explicitly stated in [6] is that *all* source variables of a relation must be bound, in order for the relation to be applied. We have formalised

this requirement by requiring *srcBinding* to be a total function on *srcVars*, in *RelationSemanticsTop*.

This seems the only reasonable approach for relation application. However it has an interesting consequence in the case of *-multiplicity references r, or indeed for any reference r with multiplicity lower bound 0. The source domain in

```
top relation R {
     checkonly domain src e : E { r = rx :
          R {}, att = attvalue };
     enforce domain trg f : F { rr = rrx :
          R1 {}, attf = attvalue };
}
```

will not match to an E instance x which has empty $x.r$, because rx cannot be bound to any element, and hence the relation R will not be applied to such x. In other cases, instances of E are copied to instances of F, and a specifier could expect that R copies $x : E$ with empty $x.r$ to a $y : F$ with empty $y.rr$, but because of the totality requirement on source bindings, no y is produced for such x.

In general, semantic issues with QVTr fall into three groups:

(1) What source bindings are possible – can relations be applied repeatedly to the same source bindings? Can source bindings be partial?
(2) The order of execution of parts of a relation – eg., whether the where clause predicates should have a defined execution order.
(3) What target bindings are possible – are new target elements created by default? Can target bindings be partially matched?

We can specify all of these issues in our semantics. For (1) we specified explicitly that repeated application of a relation to the same source binding cannot occur: according to [6], one application should be sufficient to establish the required constraints for the binding. We required that source bindings must be total.

For (2) we do not specify an order for the where clause, however there are arguments that using textual order would improve the comprehensibility of QVTr specifications [50].

For (3) we specified standard QVTr check-before-enforce rules for target binding, however other variants, such as default creation of new elements, could also be defined. Default creation semantics is used in Medini QVT, and in [51].

4 Related Work and Discussion

Kim et al. [52] have presented a formal and relatively complete approach for a model transformation. First, the two metamodels of the Object-Z formal model and the UML model (its class diagrams) are constructed. Then, in order to provide a precise, consistent and complete specification for a model transformation (with the aim of analysing syntactic and semantic issues of model elements), a bidirectional transformation from Object-Z to UML, and vice versa, is established. Finally, the proposed transformation model is represented in a practical case study.

Another formal transformation approach with the aim of capturing formal methods' benefits (i.e., correctness and completeness of formal models) by integration of UML visual notations (in fact, *class diagrams* for modeling static behavior of the real system) to the Z constructs (*schemas*) can be found in the work of Zafar and Alhumaidan [53]. To have a precise requirements analysis with support in UML design, a Z formal specification of UML class diagrams including four major relationships, i.e., association, generalization, aggregation and composition, is presented.

A very similar formal work to this research has been done by Amelunxen and Schürr [54]. Due to the incompleteness and ambiguities in the semantics of UML/MOF 2.0 metamodels and in between transformations specifically on dynamic parts, the authors want to formalize these issues using graph transformation set theoretic approaches because of the complexities of the involved associations. In contrast, the proposed formal model using the Z notation takes a more abstract view than the approach used by Amelunxen and Schürr [54].

Lano et al. [18] have presented a model transformation framework, which facilitates checking the correctness of model transformations and the related properties from different model transformation languages. The variety of transformations include the transformation style, verification of different properties regarding each language and style of the transformation.

Guerra and de Lara [55] have presented a formal semantics for the QVTr through compiling it into Petri-nets as a formal verification method. Another interesting method named game-theoretic approach has been developed by Stevens [19] for the execution semantics of QVTr transformations in check-only mode. This approach has been extended to support the enforce mode as well [56].

Greenyer and Kindler [15, 57] have implemented a TGG interpreter, which reconciles QVTr transformations with TGGs through transforming QVTr specifications to QVTc mappings. Comparing the concepts

of the declarative languages of QVT, i.e., QVTr and QVTc with TGGs, reveals many commonalities in between. For example, the relational transformation nature of the two technologies, i.e. similar structures and common underlying concepts help to transform relational QVT into TGG rules. With exploiting the formal semantics of TGGs, some semantic gaps are clarified in the QVTr and QVTc languages. Compared with our presented formalism which specifies the model transformation concepts in general and the bidirectional model transformations in particular like the QVTr transformations, the QVTc mappings are transformed to the TGG rules and interpreted by an extension of the underlying engine.

Guerra and de Lara [13] to fill the gap between the QVTr transformations and the underlying QVTc and QVTo operational counterparts presented a formal semantic for the QVTr check-only mode transformations based on algebraic specification and category theory. This formalism generalizes some details of the QVT standard by (1) formalizing the relations as bidirectional constraints and (2) providing flexibility, e.g. through passing different parameter set on calling a given relation.

Westfechtel [50] by a number of transformation cases aimed at exploring the QVTr support for the bidirectional model transformations. The used cases varies in size or the underlying source/target metamodels to challenge the functionality, solvability, variability, comprehensibility, and the semantic soundness of the QVTr standard's [6] bidirectional model transformation capabilities. This research focuses on the declarative specification of bidirectional transformations in four modes: check only/enforced modes along with the transformation direction which can be forward/backward. The model transformation executions are considered batch not incremental, i.e., the target model in the enforced mode execution is considered empty and created from scratch to enforce consistencies among the source and target models. This paper raises the semantic ambiguities and inconsistencies of the QVTr standard considering the evaluation orders of the transformation's constituent relations and their components.

In other research, Westfechtel [58] uses the well-known Persons to Families case to evaluate the defined semantics of the QVTr for bidirectional model transformations. The raised problems are (1) *imprecise change propagation*, since the QVTr language design is state-based and there is no explicit and persistent traces; (2) *unidirectional transformations*, despite the support of QVTr for the bidirectional model transformation specification, a developer has to write two separate model transformations for the forward and backward directions; (3) *noninjective mappings*, since QVTr follows the check-before-enforce semantics to specify the enforced execution mode, multiple source model elements can be mapped to the same element in the target model; and (4) *duplicate transformation*, because there is no dependency between the application of relations of each transformation, source model elements can be transformed multiple times. As discussed earlier in 3.7, these issues are resolved in our presented formalism.

Because the researches done by Stevens [19] and Bradfield and Stevens [56] take the QVTr specification issues into account and try to resolve them, here a brief discussion and comparison is made between the mentioned related semantics and the proposed formal model of this research.

- **Unresolved relations' recursion in the *when* and *where* clauses in QVTr specification.** Bradfield and Stevens [56] resolve the issue using μ-calculus [20] which has expressive power and algorithmic properties. In our approach the execution semantics is specified at a high level in terms of necessary bindings that must exist after a completed top-level relation execution (possibly including recursive invocation of non-top relations). This leaves open different approaches to defining the scheduling of *where*-invoked relations and their updates.
- **Incompleteness and inconsistencies in the specification of check-only/enforce mode, e.g., checking directions in the check-only mode as well as emerging inconsistent models after updates.** Stevens, Bradfield and Stevens [19, 56] use a game theory model involving two players, a *Verifier* and a *Refuter* which each one tries to win against the other player. First the checking problem is translated into a model-checking problem in modal μ-calculus. Then, in the check-only execution mode playing the *Verifier* tries to show that the source and target models are consistent against the model element selection of the *Refuter* move. The strength of the presented semantics is the independence from any specific metamodeling language. I.e., it is not limited to OMG MOF and OCL. Also, the formal semantic of the enforce mode has been presented by Bradfield and Stevens [56]. The target model(s) can be a fresh model or a non-empty model. One of the extensions of Bradfield and Stevens [56] to the work done by Stevens [19] is restarting the check/enforcement just after the target model update. This subject is ignored in the QVT standard [6], even though the checking of all top relations after any update will be mandatory to avoid model inconsistencies. The

current paper presents a simple formal model of the check-only and enforce modes using Z schemas which are a simpler and less specialised formalism than modal μ-calculus. Our formalism does not consider the implementation aspects of the specification of QVTr. In other words, the semantics used by [19, 56] can be utilized in the refinement of the proposed formal model. Additionally, as stated earlier, the proposed formal model of this research is limited to OMG MOF.

- **Completeness of the specification model.** Compared with the related mentioned literature [6, 19, 56], the proposed formalism of this paper is relatively complete since in addition to the execution scenarios, it covers and specifies the main concepts of model transformation including metamodels, models, transformations to name a few. Our formalism covers the main aspects of QVT-R: multi-directional execution; check-before-enforce semantics; non-persistent traces. We do not cover update-in-place semantics, however the formalism can be directly modified to address this by working in terms of a single source/target model instead of separate models.
- **Readability, understandability, and reusability aspects.** From our point of view, the specifications of this formalism are defined in a more declarative manner than previous approaches to QVT-R semantics. We have used classical logic, rathar than specialised formalisms and proof theory (such as the modal π-calculus and game theory of Stevens, Bradfield and Stevens [19, 56]) to define different execution modes of a transformation. More importantly, this paper supports a simple process to automate the construction of tool support for the MOF-based model transformations reusing the comprehensive presented formalism. Moreover, our presented semantics, compared with the existing ones like in the work of Greenyer and Kindler, Guerra and de Lara [13, 15] which requires familiarity of the reader with specialised and relatively complex algebraic semantics and category theory [13] and the extended graph structures and TGG rules [15], seems more readable and understandable as well.

5 Conclusions

After a brief introduction to the QVT standard and a survey on its related tools and languages, a formal model for the main concepts of a model transformation on a typical model transformation language using the Z notation was presented. This formalism which is based on the MOF modeling features, includes concepts such as a model, metamodel, transformation to name a few. The model transformation was specified in two execution modes: "Checkonly" and "Enforced". The obtained formal model is type checked and validated by the Z/EVES tool version 2.1 and the Alloy Analyzer tool version 4.2.

To show the applicability of the proposed formalism, the classic object-relational model transformation specification was presented in the metamodel level as an example. A discussion and comparison were made between the related work regarding formal semantics and the formal model of this research. Indeed, despite a few distinctions, the mentioned works and current research are complementing of each other.

Compared with the existing related literature [6, 19, 56], in one hand, the proposed formal model of this paper is relatively complete regarding its coverage and specification of the main concepts of model transformations. On the other hand, the specifications of this formalism are more readable and declarative than the previous approaches of the QVTr semantics.

We have used classical logic, rathar than specialised formalisms and proof theory (such as the modal π-calculus and game theory of Stevens, Bradfield and Stevens [19, 56]) to define different execution modes of a transformation. More importantly, this paper supports a simple process to automate the construction of tool support for the MOF-based model transformations reusing the comprehensive presented formalism. Moreover, our presented semantics, compared with the existing ones like in the work of Greenyer and Kindler, Guerra and de Lara [13, 15] which requires familiarity of the reader with specialised and relatively complex algebraic semantics and category theory [13] and the extended graph structures and TGG rules [15], seems more readable and understandable as well.

In the next step, based on the proposed formalism of this research, it is straightforward to develop a MOF-based model transformation supporting tool. An initial version of such a tool has been incorporated as a QVTr to UML translator in the Eclipse Agile UML toolset (https://projects.eclipse.org/ projects/-modeling.agileuml). Of course, the used formal modeling method of this research can be applied to formalize other transformation languages, domains and language concepts as well, e.g. model transformation design patterns.

References

[1] A. Boronat and J. Meseguer. An algebraic semantics for MOF. *Formal Aspects of Computing*, 22(3):269–296, May 2010. ISSN 1433-299X. doi:10.1007/s00165-009-0140-9.

[2] J. B. Tolosa, O. Sanjuán-Martínez, V. García-Díaz, B. C. P. G-Bustelo, and J. M. C. Lovelle. Towards the systematic measurement of ATL transformation models. *Software: Practice and Experience*, 41(7):789–815, 2011. doi:10.1002/spe.1033.

[3] A. P. F. Magalhaes, A. M. S. Andrade, and R. S. P. Maciel. Model Driven Transformation Development (MDTD): An Approach for Developing Model to Model Transformation. *Information and Software Technology*, 114:55–76, 2019. doi:10.1016/j.infsof.2019.06.004.

[4] J. Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, May 2005. ISSN 1619-1374. doi:10.1007/s10270-005-0079-0.

[5] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA explained, the model driven architecture: Practice and promise*. Addison-Wesley Professional, 2003.

[6] QVT. https://www.omg.org/spec/qvt/1.3/, Accessed 28 November 2019.

[7] medini QVT. http://projects.ikv.de/qvt/wiki, Accessed 28 November 2019.

[8] D. Li, X. Li, and V. Stolz. QVT-based model transformation using XSLT. *SIGSOFT Softw. Eng. Notes*, 36(1):1–8, January 2011. ISSN 0163-5948. doi:10.1145/1921532.1921563.

[9] C. Gerking and C. Heinzemann. Solving the Movie Database Case with QVTo. In *CEUR Workshop Proceedings*, volume 1305, pages 98–102, 07 2014.

[10] SmartQVT. https://sourceforge.net/projects/smartqvt, Accessed 19 July 2019.

[11] N. Kahani, M. Bagherzadeh, J. R.Cordy, J. Dingel, and D. Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, Mar 2018. ISSN 1619-1374. doi:10.1007/s10270-018-0665-6.

[12] N. Macedo and A. Cunha. Implementing QVT-R bidirectional model transformations using Alloy. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 297–311, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37057-1. doi:10.1007/978-3-642-37057-1_22.

[13] E. Guerra and J. de Lara. An algebraic semantics for qvt-relations check-only transformations. *Fundamenta Informaticae*, 114(1):73–101, 2012. doi:10.3233/FI-2011-618.

[14] D. Song, K. He, P. Liang, and W. Liu. A Formal Language for Model Transformation Specification. In *ICEIS (3)*, pages 429–433, 2005. doi:10.5220/0002546104290433.

[15] J. Greenyer and E. Kindler. Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. *Software & Systems Modeling*, 9(1):21, Jul 2009. ISSN 1619-1374. doi:10.1007/s10270-009-0121-8.

[16] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos. Engineering model transformations with transml. *Software & Systems Modeling*, 12(3):555–577, Jul 2013. ISSN 1619-1374. doi:10.1007/s10270-011-0211-2.

[17] S. Djeddai, M. Strecker, and M. Mezghiche. Integrating a Formal Development for DSLs into Meta-modeling. In Alberto Abelló, Ladjel Bellatreche, and Boualem Benatallah, editors, *Model and Data Engineering*, pages 55–66, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33609-6. doi:10.1007/978-3-642-33609-6_7.

[18] K. Lano, T. Clark, and S. Kolahdouz-Rahimi. A framework for model transformation verification. *Formal Aspects of Computing*, 27(1):193–235, Jan 2015. ISSN 1433-299X. doi:10.1007/s00165-014-0313-z.

[19] P. Stevens. A simple game-theoretic approach to checkonly QVT relations. *Software & Systems Modeling*, 12(1):175–199, Feb 2013. ISSN 1619-1374. doi:10.1007/s10270-011-0198-8.

[20] J. Bradfield and I. Walukiewicz. *The mu-calculus and Model Checking*, pages 871–919. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8_26.

[21] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., 1996.

[22] XML. https://www.omg.org/spec/xml/1.1/, Accessed 28 November 2019.

[23] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1):31–39, 2008. doi:10.1016/j.scico.2007.08.002.

[24] K. Lano. A compositional semantics of UML-RSDS. *Software & Systems Modeling*, 8(1):85–116, Feb 2009. ISSN 1619-1374. doi:10.1007/s10270-007-0064-x.

[25] K. Lano and S. Kolahdouz-Rahimi. Specification and verification of model transformations using uml-rsds. In Dominique Méry and Stephan Merz, editors, *Integrated Formal Methods*, pages 199–214, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16265-7.

[26] M. Biehl. Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, 2010.

[27] E. Jakumeit, S. Buchwald, D. Wagelaar, L. Dan, Á. Hegedüs, M. Herrmannsdörfer, T. Horn, E. Kalnina, C. Krause, K. Lano, M. Lepper, A. Rensink, L. Rose, S. Wätzoldt, and S. Mazanek. A survey and comparison of transformation tools based on the transformation tool contest. *Sci-*

ence of Computer Programming, 85:41–99, 2014. doi:10.1016/j.scico.2013.10.009.

[28] ModelMorf. http://www.mdetools.com/detail.php?toolid=61, Accessed 19 July 2019.

[29] Together. http://www.microfocus.com/products/requirements-management/together/, Accessed 19 July 2019.

[30] JQVT. https://github.com/patins1/raas4emf, Accessed 19 July 2019.

[31] E. D Willink. UMLX: A graphical transformation language for MDA. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[32] L. Freitas, J. Woodcock, and Z. Fu. Posix file store in Z/Eves: an experiment in the verified software repository. *Science of Computer Programming*, 74(4):238–257, 2009. doi:10.1016/j.scico.2008.08.001.

[33] N.A. Zafar. Formal specification and validation of railway network components using Z notation. *IET Software*, 3:312–320(8), August 2009. ISSN 1751-8806. doi:10.1049/iet-sen.2008.0082.

[34] K. Moremedi and J. A. van der Poll. Transforming formal specification constructs into diagrammatic notations. In Alfredo Cuzzocrea and Sofian Maabout, editors, *Model and Data Engineering*, pages 212–224, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-41366-7. doi:10.1007/978-3-642-41366-7_18.

[35] L. Shan and H. Zhu. A formal descriptive semantics of uml. In Shaoying Liu, Tom Maibaum, and Keijiro Araki, editors, *Formal Methods and Software Engineering*, pages 375–396, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-88194-0. doi:10.1007/978-3-540-88194-0_23.

[36] N. Amálio and F. Polack. Comparison of formalisation approaches of uml class constructs in Z and Object-Z. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B*, pages 339–358, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-44880-8. doi:10.1007/3-540-44880-2_21.

[37] K. Miloudi, Y. Amrani, and A. Ettouhami. An automated translation of UML class diagrams into a formal specification to detect UML inconsistencies. In *The Sixth International Conference on Software Engineering Advances, ICSEA*, pages 432–438, Barcelona, Spain, October 23-29 2011.

[38] A. Evans, R. France, K. Lano, and B. Rumpe. The uml as a formal modeling notation. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language. UML'98: Beyond the Notation*, pages 336–348, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48480-6. doi:10.1007/978-3-540-48480-6_26.

[39] R. Hebig, C. Seidl, T. Berger, J. K. Pedersen, and A. Wasowski. Model Transformation Languages Under a Magnifying Glass: A Controlled Experiment with Xtend, ATL, and QVT. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 445–455, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5573-5. doi:10.1145/3236024.3236046.

[40] M. Amrani, L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J. R. Cordy. A tridimensional approach for studying the formal verification of model transformations. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 921–928, April 2012. doi:10.1109/ICST.2012.197.

[41] D. Calegari and N. Szasz. Verification of model transformations: a survey of the state-of-the-art. *Electronic Notes In Theoretical Computer Science*, 292:5–25, 2013. doi:10.1016/j.entcs.2013.02.002.

[42] M. Saaltink. The Z/EVES system. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, pages 72–85, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. doi:10.1007/BFb0027284.

[43] E. G. Aydal, M. Utting, and J. Woodcock. A comparison of state-based modelling tools for model validation. In Richard F. Paige and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, pages 278–296, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-69824-1_16.

[44] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[45] S. Tarkan and V. Sazawal. Chief Chefs of Z to Alloy: Using a Kitchen Example to Teach Alloy with Z. In Jeremy Gibbons and José Nuno Oliveira, editors, *Teaching Formal Methods*, pages 72–91, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04912-5.

[46] P. Malik, L. Groves, and C. Lenihan. Translating Z to Alloy. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z*, pages 377–390, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11811-1. doi:10.1007/978-3-642-11811-1_28.

[47] J. Bezivin, F. Jouault, and D. Touzet. Principles, standards and tools for model engineering. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 28–29, June 2005. doi:10.1109/ICECCS.2005.68.

[48] D. Jackson. Alloy: A Language and Tool for Exploring Software Designs. *Commun. ACM*, 62(9):66–76, August 2019. ISSN 0001-0782. doi:10.1145/3338843. URL `http://doi.acm.org/10.1145/3338843`.

[49] X. Jia. An approach to animating Z specifications. pages 108–113. Proceedings Nineteenth Annual International Computer Software and Applications Conference (COMPSAC'95), IEEE, 1995. doi:10.1109/CMPSAC.1995.524767.

[50] B. Westfechtel. Case-based exploration of bidirectional transformations in QVT relations. *Software & Systems Modeling*, 17(3):989–1029, 2018.

[51] B. Westfechtel and T. Buchmann. Incremental Bidirectional Transformations: Comparing Declarative and Procedural Approaches Using the Families to Persons Benchmark. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 98–118. Springer, 2018.

[52] S. Kim, D. Carrington, and R. Duke. A metamodel-based transformation between UML and Object-Z. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. No.01TH8587)*, pages 112–119. Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. No.01TH8587), Sep. 2001. doi:10.1109/HCC.2001.995246.

[53] N. A. Zafar and F. Alhumaidan. Transformation of class diagrams into formal specification. *International Journal of Computer Science and Network Security*, 11(5):289–295, 2011.

[54] C. Amelunxen. Formalising model transformation rules for UML/MOF 2. *IET Software*, 2:204–222(18), June 2008. ISSN 1751-8806. doi:10.1049/iet-sen:20070076.

[55] E. Guerra and J. de Lara. Colouring: execution, debug and analysis of QVT-relations transformations through coloured petri nets. *Software & Systems Modeling*, 13(4):1447–1472, Oct 2014. ISSN 1619-1374. doi:10.1007/s10270-012-0292-6.

[56] J. Bradfield and P. Stevens. Enforcing QVT-R with mu-Calculus and Games. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 282–296, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37057-1. doi:10.1007/978-3-642-37057-1_21.

[57] J. Greenyer. A study of model transformation technologies: Reconciling TGGs with QVT. Master's thesis, University of Paderborn, 2006.

[58] B. Westfechtel. Incremental bidirectional transformations: Applying qvt relations to the families to persons benchmark. In *ENASE*, pages 39–53, 2018.

[59] J. Bézivin and F. Jouault. Using ATL for checking models. *Electronic Notes in Theoretical Computer Science*, 152:69–81, 2006. doi:10.1016/j.entcs.2006.01.015.

[60] D. Li, X. Li, and V. Stolz. QVT-based Model Transformation Using XSLT. *SIGSOFT Softw. Eng. Notes*, 36(1):1–8, January 2011. ISSN 0163-5948. doi:10.1145/1921532.1921563.

6 Appendices

6.1 A brief description of OMG MOF metamodel

Figure 5 demonstrates the modeling concepts and their relationships. The abstraction level raises from the bottom to top, i.e., from *M0* which is the used system to *M2* which is the metamodel level. The relation between levels going from the bottom to top one is instantiation/conformance. For example, each model found in layer *M1* is considered as an instance of an *M2* model, i.e. the M1's instances conform to the M2 model.

The essential concepts of a formal transformation language are defined as follows [4, 14, 59]:

- **Metamodel**: a set of concepts which define a collection of classes and their associations in a specific domain.
- **Model**: a set of entities which represents a real system abstraction.
- **Transformation**: a functional mapping which is a reflection of a relation set and an evolution of source elements to target elements.

6.2 Descriptions of the QVT-based languages and tools

mediniQVT [7] A Graphical User Interface (GUI) application and Eclipse plug-in which makes it possible to create new metamodels, new models based on these metamodels, as well as programming QVT transformations and applying desired transformations on the expected models. In fact, the mediniQVT is an engine for the execution of OMG QVT transformation standard. However, only the *Relations* language is supported by this tool. Furthermore, transformations can be executed in both directions, from the source to the target and vice versa. Here, the transformations are executed like a transaction. In other words, if all the transformation relations will be satisfied, the target model is modified, otherwise the models will be untouched.

SmartQVT [10] The first open source implementation of the QVT Operational language by the France Telecom R&D. This tool has been developed

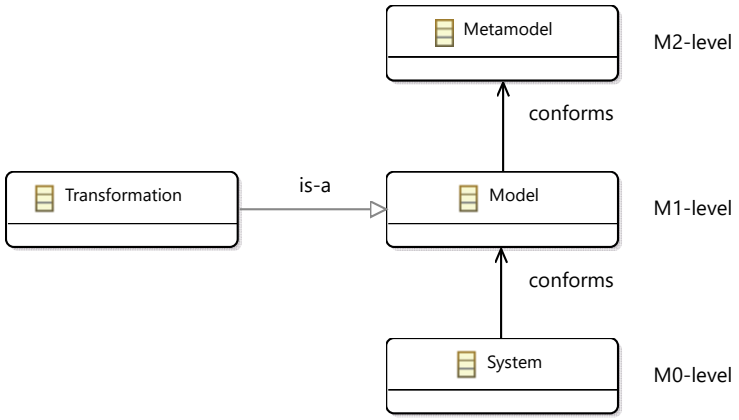

Figure 5. Modeling Level Hierarchy.

and provided as an extension plug-in for Eclipse Modeling Framework (EMF). The tool is composed of three components:

- **QVT Editor** which is used for writing QVT transformation scripts with the *.qvt* extensions. Each transformation is written as a collection of mappings from the source models to the target ones.
- **QVT Parser** which is used for representing the QVT script concrete textual syntax in terms of the QVT metamodels.
- **QVT Compiler** which compiles the QVT model and produces a Java program for executing the transformation on top of Application Programming Interfaces (APIs) which are generated by EMF. The input model of the compiler component is in XML Metadata Interchange (XMI) format and conforms to the QVT metamodel.

QVT Operational [9] Or QVTO is a partial implementation of the QVT standard. As stated in the QVT specification, complex structure transformations need imperative language features and black-box implementation. This tool aims at realization of these features. This tool is much like the SmartQVT tool (see Table 1).

QVTr-XSLT [8] It has been developed as a prototype tool to provide the transformations of QVT Relations in graphic format and to exploit from similarities between QVT Relations and XSLT in order to use the XSLT related existing tools. XSLT is a programming language to transform XML documents based on some declarative rules [60].

ModelMorf [28] An engine to implement the QVTr language. By supporting transformations in multi-directions, the same rule can be applied to map in both directions. In addition, it is possible to record the traces of the model transformation executions as well.

Together [29] Micro Focus (formerly Borland) Together is a set of Eclipse plugins which partially implements the QVT operational mappings language.

JQVT [30] A Java code generator which is based on the QVT standard specification. The tool provides features to specify relations between Java and EMF object types and creates output instances from a set of input instances by matching Java expressions used as predicates.

UMLX [31] A concrete graphical syntax aimed at complementing the QVT standard transformation capabilities.

UML-RSDS [25] This tool supports the specification and analysis of models in a subset of UML and generates executable code from them.

6.3 The Alloy model of the presented formalism

```
module qvtr/qvtr_alloy
open util/boolean
open util/ordering[VAR]
open util/relation

sig Name, Type, Value, Constraint {}

abstract sig Report {}
sig Check_is_Passed extends Report {}
sig Check_is_Failed extends Report {}
sig CREATE_REPORT extends Report {}
sig DELETE_REPORT extends Report {}

abstract sig execScenario {}
sig Checkonly extends execScenario {}
sig Enforce extends execScenario {}

sig VAR {name: Name,
  type: Type // VAR == (Name \cross Type)
}

sig Operation {
  name: Name,
  params: seq VAR,
```

```
  resultType: Type,
  precondition: Constraint,
  postcondition: Constraint,
  signature: seq Type
}
{
  signature = {i: Int, t: Type | i < #params and t =
                                    params[i].type}
}

sig Class extends Type {
  name: Name,
  properties: set VAR,
  operations: set Operation,
  invariants: set Constraint
}
{
  all p1, p2: VAR | p1 in properties and
         p2 in properties and
         p1 != p2 =>
             p1.name != p2.name
  all o1, o2: Operation | o1 in operations and
              o2 in operations and
              o1.name = o2.name and
              o1.signature = o2.signature =>
                  o1 = o2
}

sig Association {
  name: Name,
  ends: some AssocEnd
}
{
  #ends >= 2
}

sig AssocEnd {
  name: Name,
  classifier: Class,
  multiplicity: set Int
}

sig Package {
  name: Name,
  classes: set Class,
  abstracts: set Class,
  concretes: set Class,
  assocs: set Association,
  superclassOf: Class -> lone Class
}
{
  #classes >= 1
  all c1, c2: Class | c1 in classes and c2 in classes and
    c1.name = c2.name => c1 = c2
  all a: Association, e: AssocEnd | a in assocs and
                  e in a.ends =>
                      e.classifier in classes
  all a1, a2: Association | a1 in assocs and
              a2 in assocs and
              a1.name = a2.name =>
                  a1 = a2
  abstracts in classes
  concretes in classes
  abstracts & concretes = none
  abstracts + concretes = classes
  dom[superclassOf] in classes
  ran[superclassOf] in classes
  all c1, c2, c3: Class | c1 in classes and
    c2 in classes and
    c3 in classes and
    c3.superclassOf = c2 and
    c2.superclassOf = c1 =>
        c3.superclassOf = c1
  all c1, c2: Class | c1 in classes and
    c2 in classes and
    c1.superclassOf = c2 =>
        c2.superclassOf != c1
}

sig Metamodel {
  name: Name,
  pkgs: set Package,
  imports: Package -> Package
}
{
  dom[imports] in pkgs
  ran[imports] in pkgs
  all p: Package | p in pkgs =>
        not p in ran[p <: imports]
}

sig Model {
  metamodel: Metamodel,
  elements: set Value,
  predicates: set Constraint
}
{
}

fun eval[s: set Constraint]: Bool {
  True
}

fun elems[c: Constraint]: set Value {
  {v: Value}
}

sig Domain {
  name: Name,
  type: Class,
  rootVar: VAR,
  vars: set VAR,
  exeScenario: execScenario,
  model: Metamodel,
  constraints: set Constraint
}
{
  rootVar.name = name and rootVar.type = type
  rootVar in vars
  some p: Package | p in model.pkgs and
            type in p.classes
}

sig When {
  vars: set VAR,
  constraints: set Constraint
}

sig Relation {
  name: Name,
  localVars: set VAR,
  domains: set Domain,
  when: When,
  where: set Constraint,
```

```
  top: Bool,
  vars: set VAR
}
{
  all d: Domain, v: VAR | d in domains and
              v in d.vars =>
                  v in vars
  localVars + when.vars in vars
  all v: VAR | v in vars => v in localVars or
          v in when.vars or
          some d: Domain | d in domains and
              v in d.vars
  #domains >= 2
}

sig Trace {
  source: Model,
  relation: Relation,
  binding: VAR -> lone Value,
  target: Model
}
{
  ran[binding] in source.elements + target.elements
}

// The csetEval function is used to
          //  evaluate the satisfaction of a given set
// of constraints in a trace sequence
          // and a model, wrt a binding.

fun csetEval(cs: set Constraint, binding: VAR -> Value,
    ts: seq Trace, m: Model): Bool {
  True
}

sig RelationSemanticsTop extends Relation {
  direction_in: Metamodel,
  exeScenario: execScenario,
  m_s_in: Model,
  m_t: Model,
  tr: seq Trace,
  srcDoms: set Domain,
  tDom: Domain,
  srcVars: set VAR,
  scrConstraints: set Constraint,
  srcBinding: VAR -> lone Value,
  Rep_out: set Report
}
{
  top = True
  srcDoms = {d: Domain | d in domains and
            d.model != direction_in}
  tDom in domains
  tDom.model = direction_in
  all v: VAR | v in srcVars =>
            some d: Domain | d in srcDoms and
            v in d.vars or
            v in localVars or
            v in when.vars
  localVars + when.vars in srcVars
  all d: Domain, v: VAR | d in srcDoms
              and v in d.vars =>
    v in srcVars
  all c: Constraint | c in scrConstraints =>
    some d: Domain | d in srcDoms and
      c in d.constraints or
      c in when.constraints
  when.constraints in scrConstraints
  all d: Domain, c: Constraint |
    d in srcDoms =>
        c in d.constraints
  srcBinding in srcVars -> m_s_in.elements
  csetEval[scrConstraints,
            srcBinding, tr, m_s_in] = True
  #tr > 0 => m_t = tr.last.target
}

sig RelationEnforceTop extends RelationSemanticsTop {
  m_t': Model,
  tr': seq Trace
}
{
  exeScenario = Enforce
  tDom.exeScenario = Enforce
  some m: Model | m.metamodel = direction_in
    =>
    some g: VAR -> Value | dom[g] in vars and
        ran[g] in m_s_in.elements + m.elements =>
      srcVars <: g =
                srcVars <: srcBinding and
      csetEval[tDom.constraints +
                where, g, tr, m] = True and
      m_t' = m and CREATE_REPORT in Rep_out and
      some t: Trace | t.source = m_s_in and
        t.relation.name = name and
        t.binding = g and t.target = m and
        tr' = tr.add[t]
}

sig RelationEnforceNonTop extends Relation {
  direction_in: Metamodel,
  exeScenario: execScenario,
  m_s_in: Model,
  m_t, m_t': Model,
  params: seq VAR,
  srcDoms: set Domain,
  tDom: Domain,
  srcVars: set VAR,
  scrConstraints: set Constraint,
  paramBinding_in: VAR -> lone Value
}
{
  top = False
  srcDoms = {d: Domain | d in domains and
        d.model != direction_in}
  tDom in domains
  tDom.model = direction_in
  all v: VAR | v in srcVars =>
        some d: Domain | d in srcDoms and
            v in d.vars or
            v in localVars or
            v in ran[params]
  localVars + ran[params] in srcVars
  all d: Domain, v: VAR | d in
       srcDoms and v in d.vars =>
    v in srcVars
  all c: Constraint | c in scrConstraints =>
    some d: Domain | d in srcDoms
        and c in d.constraints or
            c in when.constraints
  when.constraints in scrConstraints
  all d: Domain, c: Constraint | d in srcDoms =>
        c in d.constraints
  all d: Domain | d in domains =>
```

```
        d.rootVar in ran[params]
  all v: VAR | v in ran[params] =>
    some d: Domain | d in domains and d.rootVar = v
  dom[paramBinding_in] = ran[params]
  some m: Model | m.metamodel = direction_in and
    all f: VAR -> Value | dom[f] in srcVars and
      ran[f] in m_s_in.elements + m_t.elements and
      ran[params] <: f = paramBinding_in and
      csetEval[scrConstraints, f, none ->
                        none, m_s_in] = True =>
        some g: VAR -> Value | dom[g] in vars and
          ran[g] in m_s_in.elements + m.elements =>
                  srcVars <: g =
                             srcVars <: f and
            csetEval[tDom.constraints + where,
              g, none -> none, m] = True and
    m_t' = m
}

sig RelationCheckonlyTop extends RelationSemanticsTop{}
{
  exeScenario = Checkonly
  (some m: Model |
    m.metamodel = direction_in and
    some g: VAR -> Value | dom[g] in vars and
      ran[g] in m_s_in.elements + m.elements and
        srcVars <: g =
                  srcVars <: srcBinding and
        csetEval[tDom.constraints +
                  where, g, tr, m] = True) =>
          Check_is_Passed in Rep_out and
  not (some m: Model |
    m.metamodel = direction_in and
    some g: VAR -> Value | dom[g] in vars and
        ran[g] in m_s_in.elements + m.elements and
        srcVars <: g = srcVars
                              <: srcBinding and
        csetEval[tDom.constraints + where,
              g, tr, m] = True) =>
                Check_is_Failed in Rep_out

}

sig Cleanup {
  tr: seq Trace,
  tr': seq Trace,
  m_t: Model,
  m_t': Model
}
{
  #tr > 0 => (let unusedElements =
         {v: Value | v in m_t.elements and
      (all t: Trace | t in ran[tr] and
        v not in ran[t.binding])} |
    m_t = tr.last.target and
    m_t'.elements = m_t.elements - unusedElements)
  #tr = 0 => m_t' = m_t
  tr' = tr
}

sig RelationalTransformation {
  name: Name,
  src: Metamodel,
  trg: Metamodel,
  relations: set Relation
}
{
  all r: Relation | r in relations =>
    all d: Domain | d in r.domains =>
      d.model in src + trg
}

sig TransformationExecutionStep extends
          RelationalTransformation {
  direction_in: Model,
  m_s_in: Model,
  m_t: Model,
  m_t': Model,
  tr: seq Trace,
  tr': seq Trace,
  terminated: Bool,
  terminated': Bool
}
{
  terminated = False
  direction_in in src + trg
  (some r: Relation | r in relations and
            r.RelationCheckonlyTop and
    Check_is_Passed in Rep_out =>
      r.RelationEnforceTop and terminated' = False)
  (not(some r: Relation | r in relations and
          r.RelationCheckonlyTop and
    Check_is_Passed in Rep_out) =>
      (Cleanup and terminated' = True))
}
```

Listing 2: The Alloy Model of the Proposed Formalism of the Main Characteristics of Model Transformation Languages.

6.4 A summary of the Z notation used in the presented formalism

Table 2 displays a summary of the Z notation [21] used in our presented formalism.

Table 2. A Summary of the Z Notation, X and Y Denote Sets of Items Here.

Notation	Description
$\mathbb{P}\, X$	The power set of set X.
$\mathbb{F}\, X$	The set of all finite set of set X. It must be mentioned that if X is a finite set then $\mathbb{P}\, X$ will be equal to $\mathbb{F}\, X$. $\mathbb{F}_1\, X$ will be the set of all finite set of set X excluding the empty set.
$X \leftrightarrow Y$	The power set of all pairs which are picked from the set of X and Y, respectively.
$f : X \nrightarrow Y$	A partial function declaration which maps some of the items from the domain X to the range Y.
$\mathrm{dom}\, f$	The domain of function f which is defined as $\mathrm{dom}\, f = \{x : X;\ y : Y \mid x \mapsto y \in f \bullet x\}$.
$\mathrm{ran}\, f$	The range of function f which is defined as $\mathrm{ran}\, f = \{x : X;\ y : Y \mid x \mapsto y \in f \bullet y\}$.
$f : X \rightarrow Y$	A total function declaration which maps all of the items of the domain X to the range Y. In other words, $\mathrm{dom}\, f = X$.
$\mathrm{seq}\, X$	A finite sequence of items picked from the set X which is defined as $\mathrm{seq}\, X == \{s : \mathbb{N} \nrightarrow X \mid \exists\, n : \mathbb{N} \bullet \mathrm{dom}\, s = 1 \,..\, n\}$.
$s \frown t$	Sequence s, concatenated with sequence t. As an example, if we have $s = \langle 1, 2\rangle$ and $t = \langle 5\rangle$ then $s \frown t$ will be $\langle 1, 2, 5\rangle$.
$\#s$	The number of items in the finite set/sequence s. In other words, $\#s$ returns the size of the set/sequence s.
$A \lhd R$	If A is a subset of set X and R is a relation of type $X \leftrightarrow Y$ then the restriction of the domain of relation R to the set of items contained in A, i.e. $A \lhd R$ will be defined as $\{x : X;\ y : Y \mid x \mapsto y \in R \wedge x \in A \bullet x \mapsto y\}$.

Alireza Rouhi received his B.Sc. at Kharazmi University of Tehran in September, 2000; M.Sc. at Sharif University of Technology in June, 2004; and Ph.D. at University of Isfahan in September 2017, all in Software Engineering field. He rewarded as outstanding researcher of Ph.D. students at Faculty of Computer Engineering, University of Isfahan in 2017. Currently, he is a lecturer at Azarbaijan Shahid Madani University, Tabriz, Iran. He is interested in Software Engineering in general and Model-Driven Software Engineering in particular.

Kevin Lano is Reader in Software Engineering at King's College London. He has worked for over 25 years in the fields of system specification and verification. A co-founder of the Precise UML group in 1996, he produced some of the first research on model transformation specification and verification and subsequently has developed techniques for the correct-by-construction software engineering of model transformations and for the verification of transformations. He is the author of the UML-RSDS toolset for precise model-based development.