Formalizing the main characteristics of QVT-based model transformation languages

Alireza Rouhi\textsuperscript{a,}\textsuperscript{*}, Kevin Lano\textsuperscript{b}

\textsuperscript{a}Faculty of Information Technology and Computer Engineering, Azarbaijan Shahid Madani University, Tabriz, Iran.
\textsuperscript{b}Department of Informatics, King’s College London, London, UK.

A R T I C L E I N F O.

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.

A B S T R A C T

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.

1 Introduction

Model-Driven Development (MDD) intends to develop software more productively by applying models as the main artifacts \cite{1,2,3}. 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 \cite{4}.

To realize MDD, Object-Management Group (OMG) has presented the Model-Driven Architecture (MDA) paradigm \cite{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
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 

The QVT standard/language is composed of three constituent languages: QVTr, QVT-Core (QVTo), 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 QVTt 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 $\mu$-calculus and game theory in order to investigate the execution modes of a QVT transformation in detail, i.e., checkonly/enforcement. Stevens [19], Bradfield and Walukiewicz [20] have proposed a formal semantics using $\mu$-calculus to resolve some incompleteness and ambiguities regarding the QVT standard specifications. Due to the many similarities of QVTo and Triple-Graph Grammars (TGGs) transformations, Greenyer and Kindler [15] developed an interpreter which executes QVTo mappings by transforming them to the TGG rules. Guerra and de Lara [13] to fill the gap between the QVTr transformations and the underlying QVTo and QVTr operational counterparts presented a formal semantic for the QVTr check-only mode transformations based on algebraic specification and category theory.

In 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 characteristics of the QVTr model transformation language with the following secondary 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, 21, 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
    }
    where {PackageToSchema(p, s);}
  }
  top relation AttributeToColumn {
    checkonly domain uml c: Class {
      enforce domain rdbms t: Table {
        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 {
      AttributeToColumn(g, t);
    }
  }
}

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

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 QVT language. In particular, the QVT 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 QVT by a translation to the Core language. We will aim to reconcile these two alternative semantics in our own QVT 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].

<table>
<thead>
<tr>
<th>Language/Tool</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
</tr>
</thead>
<tbody>
<tr>
<td>mediniQVT</td>
<td>D</td>
<td>E</td>
<td>P</td>
<td>M2M</td>
<td>B</td>
<td>R</td>
<td>IV</td>
<td>D</td>
</tr>
<tr>
<td>SmartQVT</td>
<td>S</td>
<td>E</td>
<td>P</td>
<td>B</td>
<td>B</td>
<td>O</td>
<td>IV</td>
<td>D</td>
</tr>
<tr>
<td>QVTo-Eclipse</td>
<td>S</td>
<td>E</td>
<td>P</td>
<td>B</td>
<td>B</td>
<td>O</td>
<td>IV</td>
<td>C</td>
</tr>
<tr>
<td>QVTr-XSLT</td>
<td>S</td>
<td>I</td>
<td>P</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>IV</td>
<td>D</td>
</tr>
<tr>
<td>ModelMorf</td>
<td>D</td>
<td>E</td>
<td>P</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>IV</td>
<td>D</td>
</tr>
<tr>
<td>Together</td>
<td>S</td>
<td>I</td>
<td>P</td>
<td>M2M</td>
<td>B</td>
<td>G</td>
<td>IV</td>
<td>C</td>
</tr>
<tr>
<td>JQVT</td>
<td>S</td>
<td>N</td>
<td>P</td>
<td>M2M</td>
<td>Ex</td>
<td>O</td>
<td>I,II</td>
<td>D</td>
</tr>
<tr>
<td>UMLX</td>
<td>S</td>
<td>I</td>
<td>P</td>
<td>M2M</td>
<td>B</td>
<td>G</td>
<td>IV</td>
<td>C</td>
</tr>
<tr>
<td>UML-RSDS</td>
<td>D</td>
<td>E</td>
<td>P</td>
<td>M2M</td>
<td>B</td>
<td>R</td>
<td>IV</td>
<td>C</td>
</tr>
</tbody>
</table>
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.
- 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, ∀ / ∃, 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.

The existing formal semantics literature 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 model transformation concepts is presented using the Z notation.

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. 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. In this research, the Z notation is used to formalize the main concepts of model transformation languages.

As stated by Stevens, one of the main drawbacks for the QVT-based tool developers is the difficulties of understanding the existing QVT standard semantics. 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 and Alloy. 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.

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 6.4.

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.

\[
\begin{align*}
\text{[Name, Type, Value, Constraint]} \\
\text{Name} & \text{ used to uniquely name the concepts such as model, package, class, domain, etc.} \\
\text{Type} & \text{ provides the data types, e.g, the primitive types such as Integer, String, etc.} \\
\text{Value} & \text{ refers to any value, either values of standard OCL primitive types such as Integer, Real, Boolean, String, or object values, instances of classes/meta-classes. Object instances } x \text{ have values } x.f \text{ for each property } (\text{feature}) f \text{ of the class of } x. \text{ The value is of the type specified for the property.} \\
\text{Constraint} & \text{ refers to any restriction on the model elements which will be evaluated as a boolean value. The constraint also can be a relation invocation in the context of the when and where clauses of a given transformation relation.} \\
\end{align*}
\]

**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 × Type, as the set of pairs of names and types by the following abbreviation:

\[
\text{Variable} \equiv \text{Name} \times \text{Type}
\]

**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.

\[
\text{Boolean ::= True | False} \\
\text{Report ::= Check\_is\_Passed | Check\_is\_Failed | CREATE\_REPORT | DELETE\_REPORT} \\
\text{execScenario ::= Checkonly | Enforce}
\]

NamedElement  This schema defines a valid named element:

- **NamedElement**
  - **name** : Name

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

- **Operation**
  - **NamedElement**
  - **params** : seq\((\text{Name} \times \text{Type})\)
  - **resultType** : Type
  - **precondition** : Constraint
  - **postcondition** : Constraint
  - **signature** : seq\((\text{Type})\)
  - \(\text{signature} = \{i : \text{dom}(\text{params}) \bullet i \mapsto \text{second}(\text{params}(i))\}\)

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:

- **Class**
  - **NamedElement**
  - **properties** : \(\text{F}(\text{Name} \times \text{Type})\)
  - **operations** : \(\text{F} \text{Operation}\)
  - **invariants** : \(\text{F} \text{Constraint}\)

\[
\forall p_1, p_2 : \text{Name} \times \text{Type} \bullet \\
\{p_1, p_2\} \subseteq \text{properties} \land p_1 \neq p_2 \Rightarrow \\
\text{first } p_1 \neq \text{first } p_2
\]

\[
\forall o_1, o_2 : \text{Operation} \bullet \\
\{o_1, o_2\} \subseteq \text{operations} \land \\
o_1.\text{name} = o_2.\text{name} \land \\
o_1.\text{signature} = o_2.\text{signature} \Rightarrow \\
o_1 = o_2
\]

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

- **Association**
  - **NamedElement ends : \(\mathbb{N}_1 \text{AssocEnd}\)**
  - \(#\text{ends} \geq 2\)

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

- **AssocEnd**
  - **NamedElement**
  - **classifier** : Class
  - **multiplicity** : \(\mathbb{N}

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

A model $m$ conforms to a given metamodel $mm$ if and only if each model element has its metaclass defined in $mm$ [37]. The statement $e \rightsquigarrow c$ 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 $x : T$, $x.f = e$, $x.f includes(e)$. 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 if the value of the feature $f$ of $x$ is equal to the value of $val$.

Likewise, a predicate

$$x.f includes(val)$$

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, $e2.g includes(e1)$
is in the predicates of a model iff \( e_1.f \rightarrow \text{includes}(e_2) \) is in the predicates.

A function
\[
elems : \text{Constraint} \rightarrow \mathbb{F}(\text{Value})
\]
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
\[
elements = \{a, b, 5, -3\}
\]
\[
predicates = \{a : A, b : A, 5 : \text{Integer}, -3 : \text{Integer}, a.att = 5, b.att = -3\}
\]

### 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 \textit{when} 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 \textit{type} variable in the \textit{Domain} schema below.

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

```java
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 \textit{Class} and \textit{Package} in the \textit{UML} metamodel, and values \( n: \text{String} \) and \( true : \text{Boolean} \).

The domain concept is formalised as:

```
Domain
---
NamedElement
type : Class
rootVar : Variable
vars : F(Variable)
execScenario : execScenario
model : Metamodel
constraints : F(Constraint)
---
rootVar = (name, type)
rootVar \in vars
\( \exists p: \text{Package} \bullet p \in \text{model.pkgs} \land \)
type \( \in p.\text{classes} \)
```

The \textit{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 \textit{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

\[
c : \text{Class}, c.\text{persistent} = \text{true},
\]
\[
p : \text{Package}, c.\text{namespace} = p, c.\text{name} = n
\]

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

The \textit{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 \( v : T, v.f = w, v.f \rightarrow \text{includes}(w) \), we also permit negations of these constraint forms in the \textit{when} clause, and tests \( v : T \) where \( T \) is abstract. The \textit{when} clause predicates will never be enforced, only evaluated.

```
When
---
vars : F(Variable)
constraints : F(Constraint)
---
```

The \textit{vars} are all variables used in the \textit{when} clause, in particular including the parameters of relation calls \( R(x, y) \). They correspond to \textit{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.

\[
\text{Relation}
\]

\[
\begin{align*}
\text{NamedElement} & : \mathbb{P}(\text{Variable}) \\
\text{localVars} & : \text{seq}((\text{Domain}) \\
\text{domains} & : \text{seq}((\text{Domain}) \\
\text{when} & : \mathbb{P}(\text{Constraint}) \\
\text{top} & : \text{Boolean} \\
\text{vars} & : \mathbb{P}(\text{Variable})
\end{align*}
\]

\[
\text{vars} = \bigcup \{ d : \text{Domain} \mid \\
\quad \quad d \in \text{dom} \text{ains} \cdot d.\text{vars} \} \cup \\
\quad \quad \text{localVars} \cup \\
\quad \quad \text{when.\text{vars}}
\]

\[
\#\text{domains} \geq 2
\]

The \text{vars} correspond to \text{R\_variable\_set} in Annex B of the QVTr v1.3 standard.

The \text{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 \( v.f \rightarrow \text{includes}(w) \) to features \( f \), for variables \( w \) (source or target) and \( v \) (target). Calls to enforce non-top relations \( R(p_1, ..., p_n) \) may also occur in the \text{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 Variables to Values:

\[
\text{Binding} ::= \text{Variable} \to \text{Value}
\]

Bindings \( g : \text{Binding} \) 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 \( \text{ran}(g) \subseteq \text{m.elements} \) for some model \( m \) associated with \( r \). If \( g((\text{name}, \text{type})) = \text{elem} \), then the type of \( \text{elem} \) in \( m \) must be consistent with \( \text{type} \), i.e., variables designated as integers must map to integer elements, etc. In addition, if \( \text{type} \) is a class in metamodel \( mm \), then \( \text{elem} \) must be of the same type in \( m \), where \( m.\text{metamodel} = mm \).

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 \( R(p_1, ..., p_n) \) 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 \( d_1, ..., d_n \) in any such occurrence are then used for \( p_1, ..., p_n \) 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 \( t \).

\[
\text{Trace}
\]

\[
\begin{align*}
\text{source} & : \text{Model} \\
\text{relation} & : \text{Relation} \\
\text{binding} & : \text{Binding} \\
\text{target} & : \text{Model}
\end{align*}
\]

\[
\text{ran}(\text{binding}) \subseteq \text{source.elements} \cup \text{target.elements}
\]

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

\[
\text{csetEval} : \mathbb{P}(\text{Constraint}) \times \text{Binding} \times \text{Trace} \times \text{Model} \to \text{Boolean}
\]

\[
\text{csetEval} = (\lambda s : \mathbb{P}(\text{Constraint}); \\
\quad f : \text{Binding}; \ ts : \text{seq Trace}; \ m : \text{Model} \cdot \\
\quad \forall c : \text{Constraint} : c \in s \cdot \\
\quad \text{eval}(c, f, ts, m) = \text{True})
\]

\[
\text{eval} : \text{Constraint} \times \text{Binding} \times \\
\text{seq Trace} \times \text{Model} \to \text{Boolean}
\]

\[
\text{csetEval}(cs, f, tr, m) \text{ evaluates to True if each constraint } c \in cs \text{ is satisfied in } m \text{ and } tr \text{ wrt } f.
\]

The following schema defines the common aspects between different execution modes (checkonly or enforce) of a top relation:
Formalizing the main characteristics of QVT-based ... —A. Rouhi and K. Lano

**RelationSemanticsTop**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Xi$</td>
<td>Relation</td>
</tr>
<tr>
<td>$\text{direction}: \text{Model}$</td>
<td>Execution direction</td>
</tr>
<tr>
<td>$\text{execScenario}: \text{execScenario}$</td>
<td>Execution scenario</td>
</tr>
<tr>
<td>$m_s, m_t: \text{Model}$</td>
<td>Source and target models</td>
</tr>
<tr>
<td>$\text{tr}: \text{seq}(\text{Trace})$</td>
<td>Trace sequence</td>
</tr>
<tr>
<td>$\text{srcDoms}: \mathbb{F}(\text{Domain})$</td>
<td>Source domains</td>
</tr>
<tr>
<td>$\text{tDom}: \text{Domain}$</td>
<td>Target domain</td>
</tr>
<tr>
<td>$\text{srcVars}: \mathbb{F}(\text{Variable})$</td>
<td>Source variables</td>
</tr>
<tr>
<td>$\text{scrConstraints}: \mathbb{F}(\text{Constraint})$</td>
<td>Source constraints</td>
</tr>
<tr>
<td>$\text{srcBinding}: \text{Variable} \rightarrow \text{Value}$</td>
<td>Source binding</td>
</tr>
<tr>
<td>$\text{Rep}!: \mathbb{F}(\text{Report})$</td>
<td>Report function</td>
</tr>
</tbody>
</table>

$\text{top} = \text{True}$

$\text{srcDoms} = \{ d : \text{Domain} \mid d \in \text{ran domains} \cdot d.\text{model} \neq \text{direction}\}$

$\text{tDom} \in \text{ran domains}$

$\text{tDom.model} = \text{direction}$

$\text{srcVars} = \bigcup \{ d : \text{srcDoms} \cdot d.\text{vars} \} \cup \text{localVars} \cup \text{when.vars}$

$\text{scrConstraints} = \bigcup \{ d : \text{srcDoms} \cdot d.\text{constraints} \} \cup \text{when.constraints}$

$\#\text{tr} > 0 \Rightarrow m_t = \text{last}(\text{tr}).\text{target}$

$\#\text{tr} = 0 \Rightarrow m_t = \emptyset$

$\text{srcBinding} \in \text{srcVars} \rightarrow m_s.\text{elements} \cup m_t.\text{elements}$

$\text{csetEval}((\text{scrConstraints}, \text{srcBinding}, \text{tr}, m_s)) = \text{True}$

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

$\neg (\exists t : \text{ran(}\text{tr}) \cdot \text{t}\text{'s relation.name} = \text{name} \land \text{srcBinding} \subseteq \text{t}\text{'s binding})$

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

An occurrence of a relation test $R(p_1, ..., p_n)$ for a top-level relation in the $\text{when}$ clause is checked against $\text{tr}$ by $\text{csetEval}((\text{scrConstraints}, \text{srcBinding}, \text{tr}, m_s))$, which includes the condition:

$\exists t : \text{ran(}\text{tr}) \cdot \text{t}\text{'s relation.name} = R\text{'s name} \land$ \begin{align*}
\text{srcBinding}(p_1) &= t\text{'s binding(d1)} \\
\vdots \\
\text{srcBinding}(p_n) &= t\text{'s binding(dn)}
\end{align*}

Where $R$ has domain sequence $d_{m_1}, ..., d_{m_n}$ with root variables $d_1, ..., d_n$.

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 $m_s$. A concept of ordering and

---

A. Rouhi and K. Lano: Formalizing the main characteristics of QVT-based ... —A. Rouhi and K. Lano

**RelationEnforceTop**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{execScenario} = \text{Enforce}$</td>
<td>Execution scenario</td>
</tr>
<tr>
<td>$\text{tDom, execScenario} = \text{Enforce}$</td>
<td>Target domain and execution scenario</td>
</tr>
<tr>
<td>$\exists m : \text{Model} \cdot m.\text{metamodel} = \text{direction} \land$ $\exists g : \text{vars} \rightarrow (m_s.\text{elements} \cup \text{elements}) \cdot$</td>
<td></td>
</tr>
<tr>
<td>$\text{srcVars} \cup g = \text{srcVars} \cup \text{srcBinding} \land$ $\text{csetEval}(\text{tDom}\text{'s constraints} \cup \text{where, g, tr, m}) = \text{True} \land$</td>
<td></td>
</tr>
<tr>
<td>$m'_t = m \land$ CREATE REPORT $\in \text{Rep}! \land$</td>
<td></td>
</tr>
<tr>
<td>$\exists t : \text{Trace} \cdot \text{t.ssource} = m_s \land \text{t.relation.name} = \text{name} \land \text{t.binding} = g \land \text{t.target} = m \land$</td>
<td></td>
</tr>
<tr>
<td>$\text{tr}' = \text{tr} \land \text{t}$</td>
<td></td>
</tr>
</tbody>
</table>

---

1 Annex B of the QVT standard 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:

\[
\text{if } t \in \text{ran}(tr) \Rightarrow \forall x : t\text{.target.elements} \Rightarrow x \in \text{ran}(t\text{.binding}) \Rightarrow x \in m'_t\text{.elements}
\]

Predicates established by preceding applications should also be preserved:

\[
\text{if } t \in \text{ran}(tr) \Rightarrow \forall c : t\text{.target.predicates} \Rightarrow \text{elems}(c) \subseteq \text{ran}(t\text{.binding}) \Rightarrow c \in m'_t\text{.predicates}
\]

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 \( c \in t\text{Dom.constrains} \) is either of form \( v.f = w \) for variables \( v \) and \( w \) and feature \( f \), or of form \( v : T \) or \( v.f \rightarrow \text{includes}(w) \).
3. In the case of \( v : T \) for type \( T \):
   - If there is no element of \( m \) with constraint \( e : T \) in \( m\text{.predicates} \), add a new \( e \) to \( m\text{.elements} \), with the constraint \( e : T \) and extend \( g \) by the binding \( v \mapsto e \).
   - If there is an existing \( e \) in \( m\text{.elements} \) which satisfies all of the required constraints of \( v \), extend \( g \) by the binding \( v \mapsto e \).
   - Otherwise, create a new \( e \) and proceed as in case (a).
4. In the case of \( v.f = w \), if there are existing elements \( e_1, e_2 \) with \( g(v) = e_1, g(w) = e_2 \):
   - If \( e_1.f = e_2 \in m\text{.predicates} \) then no updates are needed

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:

\[
\text{top relation } R \{
\text{checkonly domain } ax : A \{ \text{att} = \text{attvalue} \};
\text{enforce domain } bx : B \{ \text{attb} = \text{attvalue} \};
\}
\]

Consider the case of executing a transformation which consists of this single rule, on an initial MM1 model with elements \( \{a_1, a_2, 1\} \), and constraints \( \{a_1 : A, a_2 : A, 1 : Integer, a_1\text{.att} = 1, a_2\text{.att} = 1\} \). This is \( m_t \)? for the first application of \( R \), and the srcBinding for this application could be:

\[
\{ ax \mapsto a_1, \text{attvalue} \mapsto 1 \}
\]

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 \( bx\text{.attb} = \text{attvalue} \). Since \( m \) is initially empty, by clause 3(a), a new element \( b_1 \) is added to \( m\text{.elements} \), constraint \( b_1 : B \) added to \( m\text{.predicates} \), and binding \( bx \mapsto b_1 \) added to \( g \).

For constraint \( bx\text{.attb} = \text{attvalue} \), clause 4(c) applies, and \( m\text{.predicates} \) is extended with a new constraint \( b_1\text{.attb} = 1 \). Thus the resulting target model \( m'_t \) has

\[
\begin{array}{c|c}
A & B \\
\hline
\text{att} : \text{Integer} & \text{attb} : \text{Integer}
\end{array}
\]
Formalizing the main characteristics of QVT-based . . . —A. Rouhi and K. Lano

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 $m'_1$. 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 $m_1$ 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 $w_1; ...; w$, then intermediate target models $m'_i$ are created where $m'_1$ is the $m$ produced by considering the target domain constraints as described above, and $m'_{i+1}$ is produced from $m'_i$ by satisfying constraint $w_i$.

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 $m_0$ input. It returns an updated $m$ as its $m_0'$ 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:

\[
\begin{align*}
\Xi_{Relation} \\
\text{direction? : Metamodel} \\
\text{exceScenario : exceScenario} \\
\text{m}_0 : \text{Model} \\
\text{m}_1, \text{m}'_1 : \text{Model} \\
\text{params : seq(Variable)} \\
\text{srcDom : } F(Domain) \\
tDom : \text{Domain} \\
\text{srcVars : } F(\text{Variable}) \\
\text{scrConstraints : } F(\text{Constraint}) \\
\text{paramBinding? : Variable } \mapsto \text{Value}
\end{align*}
\]

\[
\begin{align*}
top = \text{False} \\
\text{srcDom} = \{d : Domain \mid d \in \text{ran domains} \land d.model \neq \text{direction?}\} \\
tDom \in \text{ran domains} \\
tDom.\text{model} = \text{direction?} \\
\text{srcVars} = \bigcup\{d : \text{srcDom} \land d.\text{vars} \land \text{ran}(\text{params})\} \\
\text{scrConstraints} = \bigcup\{d : \text{scrDom} \land d.\text{constraints}\} \\
\forall d : \text{Domain} \land d.\text{rootVar} \in \text{ran params} \\
\forall r : \text{dom params} \\
\text{params}(i) = \text{domains}(i).\text{rootVar} \\
\text{dom(paramBinding?) = ran(params)} \\
\exists m : \text{Model} \land m.\text{metamodel} = \text{direction?} \land \\
\forall f : \text{srcVars} \rightarrow (m_0.\text{elements} \land m_0.elems) \land \\
\text{ran(params)} \land f = \text{paramBinding?} \land \\
\text{csetEval}(\text{scrConstraints}, f, \langle \rangle, \{\}) = \text{True} \Rightarrow \\
\exists g : \text{vars} \rightarrow (m_0.\text{elements} \land m.elems) \land \\
\text{srcVars} \land g = \text{srcVars} \land f \land \\
\text{csetEval}(t\text{Dom}.\text{constraints} \land \text{where}, g, \{\}, m) = \text{True} \land \\
m'_1 = m
\end{align*}
\]

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:
The model predicates of unusedElements in binding in the trace targets: elements which do not appear in the range of any formation execution, which removes target model elements which have not been explicitly deleted 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:

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

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 [18]. 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 [14].
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[4] 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 = Class,
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 = ((), ()),
where = ⟨⟩,
top = true)

Relation ::= (name = AttributeToColumn,
localVars = ∅,
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})]

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
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 $m_2$, for a rule execution in $\text{RelationEnforceTop}$ and $\text{RelationEnforceNonTop}$: 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 $e1, e2, \ldots$ are required to exist by constraints such as $v.r \rightarrow \text{includes}(w)$, 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 not explicitly stated in [5] is that all source variables of a relation must be bound, in order for the relation to be applied. We have formalised...
Formalizing the main characteristics of QVT-based ... —A. Rouhi and K. Lano

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.

Greenyer and Kindler [15, 57] have implemented a TGG interpreter, which reconciles QVT transformations with TGGs through transforming QVT 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 [58] 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 [57] these issues are resolved in our presented formalism.

Because the research 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 [2], 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 $\mu$-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, rather than specialised formalisms and proof theory (such as the modal $\pi$-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, rather than specialised formalisms and proof theory (such as the modal $\pi$-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


Formalizing the main characteristics of QVT-based... —A. Rouhi and K. Lano


6 Appendices

6.1 A brief description of OMG MOF metamodel

Figure 6 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...
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 QVT 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).

**QVT-XML** [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 QVT language. By supporting transformations in multidirections, 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

```alloy
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 \cross Type}  

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

**Figure 5. Modeling Level Hierarchy.**
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
}
{ all c1, c2, c3: Class | c1 in classes and
  c2 in classes and
  c3 in classes and
  c3.superclassOf = c2 and
  c2.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,
  class: Class
}
}
Formalizing the main characteristics of QVT-based... —A. Rouhi and K. Lano

\[
\begin{align*}
\text{top: } \text{Bool}, \\
\text{vars: set } \text{VAR} \\
\end{align*}
\]

\[
\begin{align*}
\{ \\
\quad \text{all } d: \text{Domain}, v: \text{VAR} \mid d \text{ in domains and } \\
\quad \quad v \text{ in } d.\text{vars} \Rightarrow \\
\quad \quad v \text{ in vars} \\
\quad \text{localVars + when.vars in vars} \\
\quad \text{all } v: \text{VAR} \mid v \text{ in vars} \Rightarrow v \text{ in localVars or } \\
\quad \quad v \text{ in when.vars or } \\
\quad \quad \text{some } d: \text{Domain} \mid d \text{ in domains and } \\
\quad \quad v \text{ in } d.\text{vars} \\
\quad \#\text{domains} \geq 2
\}
\]

\[
\begin{align*}
\text{sig Trace} \\
\quad \text{source: Model,} \\
\quad \text{relation: Relation,} \\
\quad \text{binding: } \text{VAR} \rightarrow \text{lone Value,} \\
\quad \text{target: Model} \\
\}
\]

\[
\begin{align*}
\{ \\
\quad \text{ran[binding]} \text{ in source.elements + target.elements} \\
\}
\]

// 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.

\[
\begin{align*}
\text{fun csetEval(cs: set Constraint, binding: VAR \rightarrow Value,} \\
\quad ts: \text{seq Trace, m: Model): Bool} \\
\quad \text{True} \\
\}
\]

\[
\begin{align*}
\text{sig RelationSemanticsTop extends Relation} \\
\quad \text{direction_in: Metamodel,} \\
\quad \text{exeScenario: execScenario,} \\
\quad \text{m_s_in: Model,} \\
\quad \text{m_t: Model,} \\
\quad \text{tr: seq Trace,} \\
\quad \text{srcDoms: set Domain,} \\
\quad \text{tDom: Domain,} \\
\quad \text{srcVars: set VAR,} \\
\quad \text{scrConstraints: set Constraint,} \\
\quad \text{srcBinding: VAR \rightarrow lone Value,} \\
\quad \text{Rep_out: set Report} \\
\}
\]

\[
\begin{align*}
\{ \\
\quad \text{top = True} \\
\quad \text{srcDoms = } \{d: \text{Domain} \mid d \text{ in domains and } \\
\quad \quad d.\text{model} \neq \text{direction_in} \} \\
\quad \text{tDom in domains} \\
\quad \text{tDom.model = direction_in} \\
\quad \text{all } v: \text{VAR} \mid v \text{ in srcVars} \Rightarrow \\
\quad \quad \text{some } d: \text{Domain} \mid d \text{ in srcDoms and } \\
\quad \quad v \text{ in d.vars or } \\
\quad \quad v \text{ in localVars or } \\
\quad \quad v \text{ in when.vars} \\
\quad \text{localVars + when.vars in srcVars} \\
\quad \text{all } d: \text{Domain}, v: \text{VAR} \mid d \text{ in srcDoms} \\
\quad \quad \text{and } v \text{ in d.vars} \\
\quad \quad v \text{ in srcVars} \\
\quad \text{all } c: \text{Constraint} \mid c \text{ in scrConstraints} \Rightarrow \\
\quad \quad \text{some } d: \text{Domain} \mid d \text{ in srcDoms and } \\
\quad \quad c \text{ in } d.\text{constraints or } \\
\quad \quad c \text{ in when.constraints} \\
\}
\]

\[
\begin{align*}
\quad \text{srcBinding in srcVars \rightarrow m_s_in.elements} \\
\quad \text{csetEval[scrConstraints,} \\
\quad \quad \text{srcBinding, tr, m_s_in] = True} \\
\quad \#tr > 0 \Rightarrow m_t = tr.last.target \\
\}
\]

\[
\begin{align*}
\text{sig RelationEnforceTop extends RelationSemanticsTop} \\
\quad \text{m_t': Model,} \\
\quad \text{tr': seq Trace} \\
\}
\]

\[
\begin{align*}
\{ \\
\quad \text{exeScenario = Enforce} \\
\quad \text{tDom.exeScenario = Enforce} \\
\quad \text{some } m: \text{Model} \mid m.\text{metamodel} = \text{direction_in} \Rightarrow \\
\quad \quad \text{some } g: \text{VAR} \rightarrow \text{Value} \mid \text{dom}[g] \text{ in vars and } \\
\quad \quad \text{ran}[g] \text{ in m_s_in.elements + m.elements} \Rightarrow \\
\quad \quad \text{srcVars < g = } \\
\quad \quad \text{srcVars < srcBinding and} \\
\quad \quad \text{csetEval[tDom.constraints +} \\
\quad \quad \quad \text{where, g, tr, m] = True and} \\
\quad \quad \text{m_t' = m and CREATE_REPORT in Rep_out and} \\
\quad \quad \text{some } t: \text{Trace} \mid t.\text{source} = m_s_in and} \\
\quad \quad \text{t.relation.name = name and} \\
\quad \quad \text{t.binding = g and t.target = m and} \\
\quad \quad \text{tr' = tr.add[t]}
\}
\]

\[
\begin{align*}
\text{sig RelationEnforceNonTop extends Relation} \\
\quad \text{direction_in: Metamodel,} \\
\quad \text{exeScenario: execScenario,} \\
\quad \text{m_s_in: Model,} \\
\quad \text{m_t, m_t': Model,} \\
\quad \text{params: seq VAR,} \\
\quad \text{srcDoms: set Domain,} \\
\quad \text{tDom: Domain,} \\
\quad \text{srcVars: set VAR,} \\
\quad \text{scrConstraints: set Constraint,} \\
\quad \text{paramBinding_in: VAR \rightarrow lone Value} \\
\}
\]

\[
\begin{align*}
\{ \\
\quad \text{top = False} \\
\quad \text{srcDoms = } \{d: \text{Domain} \mid d \text{ in domains and } \\
\quad \quad d.\text{model} \neq \text{direction_in} \} \\
\quad \text{tDom in domains} \\
\quad \text{tDom.model = direction_in} \\
\quad \text{all } v: \text{VAR} \mid v \text{ in srcVars} \Rightarrow \\
\quad \quad \text{some } d: \text{Domain} \mid d \text{ in srcDoms and } \\
\quad \quad v \text{ in d.vars or } \\
\quad \quad v \text{ in localVars or } \\
\quad \quad v \text{ in ran[params]} \\
\quad \text{localVars + ran[params] in srcVars} \\
\quad \text{all } d: \text{Domain}, v: \text{VAR} \mid d \text{ in srcDoms} \\
\quad \quad \text{and } v \text{ in d.vars} \\
\quad \quad v \text{ in srcVars} \\
\quad \text{all } c: \text{Constraint} \mid c \text{ in scrConstraints} \Rightarrow \\
\quad \quad \text{some } d: \text{Domain} \mid d \text{ in srcDoms and } \\
\quad \quad c \text{ in } d.\text{constraints or } \\
\quad \quad c \text{ in when.constraints} \\
\quad \text{when.constraints in scrConstraints} \\
\quad \text{all } d: \text{Domain}, c: \text{Constraint} \mid d \text{ in srcDoms} \Rightarrow \\
\quad \quad c \text{ in } d.\text{constraints} \\
\quad \text{all } d: \text{Domain} \mid d \text{ in domains} \Rightarrow
\}
\]
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.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{P} X )</td>
<td>The power set of set ( X ).</td>
</tr>
<tr>
<td>( \mathbb{F} X )</td>
<td>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.</td>
</tr>
<tr>
<td>( X \leftrightarrow Y )</td>
<td>The power set of all pairs which are picked from the set of ( X ) and ( Y ), respectively.</td>
</tr>
<tr>
<td>( f : X \rightarrow Y )</td>
<td>A partial function declaration which maps some of the items from the domain ( X ) to the range ( Y ).</td>
</tr>
<tr>
<td>( \text{dom} f )</td>
<td>The domain of function ( f ) which is defined as ( \text{dom} f = { x : X ; y : Y</td>
</tr>
<tr>
<td>( \text{ran} f )</td>
<td>The range of function ( f ) which is defined as ( \text{ran} f = { x : X ; y : Y</td>
</tr>
<tr>
<td>( f : X \rightarrow Y )</td>
<td>A total function declaration which maps all of the items of the domain ( X ) to the range ( Y ). In other words, ( \text{dom} f = X ).</td>
</tr>
<tr>
<td>( \text{seq} X )</td>
<td>A finite sequence of items picked from the set ( X ) which is defined as ( \text{seq} X = { s : \mathbb{N} \Rightarrow X</td>
</tr>
<tr>
<td>( s \triangleq t )</td>
<td>Sequence ( s ), concatenated with sequence ( t ). As an example, if we have ( s = \langle 1, 2 \rangle ) and ( t = \langle 5 \rangle ) then ( s \triangleq t ) will be ( \langle 1, 2, 5 \rangle ).</td>
</tr>
<tr>
<td>( # s )</td>
<td>The number of items in the finite set/sequence ( s ). In other words, ( # s ) returns the size of the set/sequence ( s ).</td>
</tr>
<tr>
<td>( A \triangleleft R )</td>
<td>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 \triangleleft R ) will be defined as ( { x : X ; y : Y</td>
</tr>
</tbody>
</table>