Implementing QVT-R via semantic interpretation in UML-RSDS

K. Lano1,2 · S. Kolahdouz-Rahimi1,2

Received: 13 January 2020 / Revised: 20 July 2020 / Accepted: 11 August 2020 / Published online: 24 September 2020 © The Author(s) 2020

Abstract
The QVT-Relations (QVT-R) model transformation language is an OMG standard notation for model transformation specification. It is highly declarative and supports (in principle) bidirectional (bx) transformation specification. However, there are many unclear or unsatisfactory aspects to its semantics, which is not precisely defined in the standard. UML-RSDS is an executable subset of UML and OCL. It has a precise mathematical semantics and criteria for ensuring correctness of applications (including model transformations) by construction. There is extensive tool support for verification and for production of 3GL code in multiple languages (Java, C#, C++, C, Swift and Python). In this paper, we define a translation from QVT-R into UML-RSDS, which provides a logically oriented semantics for QVT-R, aligned with the RelToCore mapping semantics in the QVT standard. The translation includes variation points to enable specialised semantics to be selected in particular transformation cases. The translation provides a basis for verification and static analysis of QVT-R specifications and also enables the production of efficient code implementations of QVT-R specifications. We evaluate the approach by applying it to solve benchmark examples of bx.

Keywords Model transformations · QVT-Relations · UML-RSDS · Model transformation semantics · Model transformation tools

1 Introduction
Model transformations (MT) are used in model-driven engineering (MDE) to map data of a source model src to a target model trg, where the models conform to particular source and target metamodels/languages SL and TL. Transformation specifications (e.g. in QVT-R, ATL or UML-RSDS) typically consist of a collection of transformation rules, each of which is concerned with mapping source elements of one or more SL classes to target elements of one or more TL classes.

A unidirectional transformation τ can only be executed in one direction (from SL models to TL models, or from TL models to SL models) whilst a bidirectional transformation (or bx) has both forward τ→ and reverse τ← mappings, derived from the same transformation specification τ. The reverse mapping operates on models trg of TL to produce models src of SL. Bidirectional transformations are not necessarily bijective as functions.

Two execution modes can be distinguished for either forward or reverse transformation directions:

- **Batch-mode** execution, where an empty trg (or src) model is populated from a src (or trg) model in one complete execution.

- **Incremental-mode** execution, where incremental changes to a src (or trg) model are propagated to changes to an already populated trg (or src) model. Changes can be: creation/deletion of elements; reassignment of 1-multiplicity features; addition/removal of elements from other features. In this paper, we address source-to-target incremental change propagation in the sense of [10].

Apart from models, incremental-mode execution could also make use of persistent traces, which record src → trg correspondences established by previous executions of the transformation.

The QVT-Relations (QVT-R) language is an OMG standard for model transformation specification. The latest cur-
The current release is version 1.3, defined in [30]. The language was intended to support the declarative specification of model transformations, avoiding imperative constructs, and supporting change propagation from one model to another, and bi- (or multi-) directional interpretation of transformations.

Although QVT-R has been widely used in research, there remain several limitations with the language which prevent wider industrial adoption:

- Incompleteness in the semantics makes it difficult to verify transformations, or to systematically design bx [36].
- Unclear semantics for update-in-place transformations makes it difficult to define and use such transformations. The combination of bx and update-in-place execution has not been developed [41].
- Tool support is incomplete, with the only mature tool, Medini QVT [11], no longer actively maintained. The tool uses a restricted version of QVT-R, with a variant semantics which has not been formalised.

In this paper, we aim to address these defects by providing a translation from QVT-R into the UML-RSDS formalism [18], which is a subset of UML with a formal semantics and extensive tool support. UML-RSDS directly supports transformation analysis and update-in-place execution of transformations, and efficient execution via code generation in 3GLs. The translation has itself been formalised in UML-RSDS. UML-RSDS is implemented in the Eclipse Agile UML tools (https://projects.eclipse.org/projects/modeling.agileuml). A guide to using the QVT2UMLRSRDS translator is at [26].

The overall process which we use to analyse and implement QVT-R transformations is illustrated in Fig. 1. All of the steps are automated, although there may be user choices to be made in the synthesis of designs. The present paper concerns the semantic derivation and analysis steps; the design synthesis and code generation steps have been previously described [18,19,22].

Figure 2 shows the subset of the QVT-R metamodel which we address in this paper. Black-box operations are not included, nor are collection templates in target domains or the ‘opposite’ navigation mechanism. The computational part of a QVT-R specification (a relational transformation) consists of a set of rule definitions (termed relations) and a set of query operation definitions. The rules and queries have distinct names:

\[
\begin{align*}
\text{rule} & \rightarrow \text{isUnique}(\text{name}) \\
\text{helpers} & \rightarrow \text{isUnique}(\text{name}) \\
\text{rule.name} & \rightarrow \text{intersection}(\text{helpers.name}) = \text{Set}() 
\end{align*}
\]

Rules may be top level, in which case they are executed on all matching source elements for which they are enabled, or non-top level, when they can only execute if explicitly invoked from a rule.

For example, considering the UML2C specification of a UML to ANSI C code generator [3] (Fig. 3 shows simplified extracts of the metamodels of this system), a simple rule that

\[
\begin{align*}
\text{rule} & \rightarrow \text{isUnique}(\text{name}) \\
\text{helpers} & \rightarrow \text{isUnique}(\text{name}) \\
\text{rule.name} & \rightarrow \text{intersection}(\text{helpers.name}) = \text{Set}() 
\end{align*}
\]
maps a UML model instance to a C program instance (and vice versa) could be written as a top relation

```java
public interface QVT2UMLRSDS {
    // relations
    TopRelationModel2Program Model2Program {
        enforce domain design u : UMLModel {
            name = n;
        }
        enforce domain design C p : CProgram {
            name = n;
        }
    }
}
```

A QVT-R rule has a sequence of domains (Rule::domain in Fig. 2). Each domain of a relation represents a source or target element of a specific type, e.g. `u:UMLModel`, from the `typedModel` of the domain, in this case the `design` model. This element is represented by the `root variable` of the domain (`RelationDomain::rootVariable` in Fig. 2). The remainder of the domain is a `template pattern` (`RelationDomain::pattern.templateExpression`) which matches and constrains the data of the root element. Templates consist of specialised forms of expression that specify individual source or target elements, `ObjectTemplateExp`, or collections of source elements, `CollectionTemplateExp`. Non-top relations may also have `primitive` domains, which only have a `root variable`, and an empty template expression. Primitive domains are used to pass in non-object parameter values. The sequence `domain.rootVariable` of domain `root variable`s is the (input) parameters of the non-top relation.

In `Model2Program`, the domain template expression for `u` introduces an auxiliary local variable `n` of the relation and assigns this the value of `u.name`. Such local variables (included in `Pattern :: bindsTo`) can also be explicitly declared in the relation:

```java
public interface QVT2UMLRSDS {
    // relations
    TopRelationModel2Program Model2Program {
        enforce domain design u : UMLModel {
            name = n;
        }
        enforce domain design C p : CProgram {
            name = n;
        }
    }
}
```

In this paper, we will only use implicit declarations for such local variables.

### 2.2 QVT-R relation semantics

The data of an `enforce` domain (a domain with `isEnforceable = true`) can be modified by application of the relation, whilst data of a `checkonly` or `primitive` domain can only be queried. Transformations have a directionality—they may be executed in the direction of any one of their parameters (typed models), in which case all their rules are also executed in this direction. This means that relation domains with this (target) model can have their data modified (in the case of `enforce` domains) using data read from domains with other
models. Elements within the target model can also be created or deleted.

Thus, for the \texttt{Model2Program} relation, if it is executed in the direction of an initially empty C model, the relation creates \texttt{p} as a \texttt{CProgram} instance and sets \texttt{p.name} = \texttt{n}, where \texttt{n} = \texttt{u.name}, the name of the \texttt{UMLModel} instance. If executed in the design direction, the relation would instead create \texttt{u} : \texttt{UMLModel} instances from \texttt{p} : \texttt{CProgram} instances and copy \texttt{p.name}’s value to \texttt{u.name}.

An important mechanism in QVT-R is \texttt{check-before-enforce} semantics: if the constraints of a relation already hold for some target elements, relative to specific source elements, then the relation execution binds some such target elements to the target domain variables, instead of creating new elements. Thus, executing \texttt{Model2Program} in the C direction, with a non-empty C model, the creation of a new \texttt{CProgram} only occurs if there is not already a \texttt{CProgram} instance satisfying the required property \texttt{name} = \texttt{u.name}, likewise for execution in the design (UML) direction.

A more complex relation defines the mapping of UML class types to C pointer types together with a C struct type:

$$
\text{top relation Class2CPointerType} \\
\{ \text{enforce domain design } e : \text{Entity} \\
\{ \text{typeId} = t, \text{name} = n \} ; \\
\text{enforce domain C p : CPointerType} \\
\{ \text{cTypeId} = t, \text{pointsTo} = c : \text{CStruct} \\
\{ \text{name} = n, \text{cTypeId} = n \} \} ; \\
\}\}
$$

The \texttt{typeId} and \texttt{cTypeId} features are unique identifiers for the UML and C type occurrences (i.e. keys in terms of QVT-R, Sect. 2.4). Executed in the C direction, for each \texttt{e} : \texttt{Entity}, both a \texttt{CPointerType} instance and a linked \texttt{CStruct} instance are potentially created. In general, any number of linked elements may be defined in a domain template. We describe this situation as (vertical) \texttt{entity splitting} [20], and it is common in refinement transformations.

The QVT-R template expressions describe the data of model elements. They can also be interpreted as standard OCL expressions. For example, the above domains have the OCL interpretations

$$
e : \text{Entity and } t = e.typeId \text{ and } n = e.name$$

and

$$p : \text{CPointerType and } \text{p.cTypeId} = t \text{ and } c : \text{CStruct and } \text{p.pointsTo} = c \text{ and } c.name = n \text{ and } c.cTypeId = n$$

Domains may have additional condition expressions (\texttt{TemplateExp :: where} in Fig. 2), which are conjoined with the logical interpretation of the template expression. For example, we could write:

$$
\text{top relation Class2CPointerTypeV2} \\
\{ \text{enforce domain design } e : \text{Entity} \\
\{ \text{typeId} = t, \text{name} = n \} ; \\
\text{e.isAbstract} = \text{false} \} ; \\
\text{enforce domain C p : CPointerType} \\
\{ \text{cTypeId} = t, \text{pointsTo} = c : \text{CStruct} \\
\{ \text{name} = n, \text{cTypeId} = n \} \} ; \\
\}\}
$$
In the C direction, this additional condition restricts the mapping to concrete UML classes. In the design direction, the condition assigns \( e.\text{isAbstract} = \text{false} \) for the target \( e : \text{Entity} \).

### 2.3 QVT-R relation dependencies

Top relations may have dependencies on other relations, in cases where a relation \( R \) can only be applied to an element \( x \) if another relation \( P \) has been previously applied to a linked element \( y \). Such dependencies are specified in a \( \text{when} \{ P(y) \} \) clause of \( R \) (Relation :: when in Fig. 2). For example, consider the mapping of UML properties to C struct members, omitting the mapping of property types:

```plaintext
// top relation Property2CMember
{ enforce domain design p : Property
  { owner = e : Entity {}, name = n, isUnique = k };  
enforce domain C m : CMember
  { memberOf = s : CStruct { }, name = n, isKey = k };  
  when { Class2CStruct(e,s) } }
```

The \( s : \text{CStruct} \) must already have been created for \( e = p.\text{owner} \), before \( \text{Property2CMember} \) can be applied to create \( m : \text{CMember} \) for \( p \). The relation \( \text{Class2CStruct} \) is:

```plaintext
// top relation Class2CStruct
{ enforce domain design e : Entity
  { name = n };  
enforce domain C c : CStruct
  { name = n, ctypeId = n }; }
```

Using this relation, \( \text{Class2CPointerType} \) can be respecified as:

```plaintext
// top relation Class2CPointerTypeV3
{ enforce domain design e : Entity
  { typeId = t };  
enforce domain C p : CPointerType
  { ctypeId = t, pointsTo = c : CStruct { } };  
  when { Class2CStruct(e,c) } }
```

Rules may inherit from other rules (Rule :: overrides in Fig. 2): an abstract rule

```plaintext
abstract top relation S2T
{ enforce domain src s : S { ... };  
enforce domain trg t : T { ... }; }
```

can be overridden by an abstract or concrete relation with more specific source/target types \( S1/T1 \) and/or source domain condition and/or when condition:

```plaintext
// top relation S12T overrides S2T
{ enforce domain src s : S1 { ... } { P };  
enforce domain trg t : T1 { ... }; }
```

The semantics of \textit{overrides} is not specified in [30]; we provide a semantics in Sect. 5.2 and “Appendix D”. We add the restriction that concrete rules should have concrete domain types \( S1, T1 \) in their direction of execution (because elements of these classes may need to be instantiated by the rule execution).

### 2.4 Specialised control of transformation behaviour

So far we have only used 1-multiplicity attribute features (such as \( \text{name} : \text{String} \)) or 1-multiplicity reference features (such as \( \text{owner} : \text{Entity} \)) in domain templates. Semantic problems begin to appear when optional features are used in domains, i.e. features \( f \) with multiplicity \( 0..n \) or \( * \). For example, the following top relation could seem to be a valid alternative way of mapping classes to structs and attributes to members in a single rule:

```plaintext
// top relation Class2CStructV2
{ enforce domain design e : Entity
  { name = n, ownedAttribute = p : Property { name = n1, isUnique = k } };  
enforce domain C c : CStruct
  { name = n, ctypeId = n, members = m : CMember { name = n1, isKey = k } }; }
```

However, there are two problems with this:

1. If \( e.\text{ownedAttribute} \) is empty, then no \( p : \text{Property} \) can match the \( e \) template, and the relation will not be applied to \( e \). In other words, classes without owned properties will not be mapped to \( C \) at all.
2. Even for classes with properties, the logic of the rule is that for every pair \((e, p)\) of a class \( e \) and an owned property \( p \), there should exist a pair \((c, m)\) of a struct and a member. In the absence of key declarations, there is no obligation that the same \( c \) is chosen for different \( p \) within one class \( e \).
Notice that check-before-enforce does not avoid problem 2: the C domain will be executed to create c and m in any case where there is not already both a CStruct and a CMember with the required logical properties relative to e and p.

One technique that QVT-R provides to address the second problem are keys: a transformation can specify that certain features or feature combinations uniquely identify elements of certain classes. Multiple elements with the same key feature values/combinations are not permitted, and target elements are looked up by key and updated when they already exist, instead of being created. In the above example, key specifications

key Entity { name };  
key CStruct { ctypeId };  

would resolve the second problem, because the same (unique) c : CStruct with c.ctypeId = e.name would be chosen for each p : e.ownedAttribute from the source domain.

Compound keys can be specified as, e.g.

key CMember { memberOf, name };  

Another means to restrict execution choices is to use non-top relations, which are relations which can be invoked in the where clauses of relations (Rule :: where in Fig. 2). For example,

relation NonTopProperty2CMember  
{ enforce domain design e : Entity  
  { ownedAttribute = p : Property  
    { name = n, isUnique = k }  
  };  
  enforce domain C c : CStruct  
  { members = m : CMember { name = n, isKey = k } }  
}

is a non-top relation with input parameters e : Entity, c : CStruct corresponding to its domain root variables. These parameters must be bound to specific elements when a call is made to the relation, e.g. as:

top relation Class2CStructV3  
{ enforce domain design e : Entity  
  { name = n }  
};  
  enforce domain C c : CStruct  
  { name = n, c.ctypeId = n }  
};  
  where { NonTopProperty2CMember(e,c) }  
}

The effect of NonTopProperty2CMember is that for each pair (e, c) of related class e and C struct c, for all properties p of e, a corresponding C member m is added to c. This resolves problems (1) and (2) above.

Problems (1) and (2) arise because matching of domain patterns in QVT-R is performed in an element-by-element manner, i.e. there is an implicit quantifier e.owned Attribute→ for All(p)... over the Class2CStructV2 relation. This is in contrast to languages such as ATL or UML-RSDS, where assignments can be used to set the values of collection-valued features of target elements in a single step, based on the collection values of source element features.

where clauses can also contain assignments to features of elements, e.g. a non-standard way of writing Model2Program could be:

top relation Model2ProgramV2  
{ enforce domain design u : UMLModel { }  
};  
  enforce domain C p : CProgram { }  
};  
  where  
  { p.name = u.name;  
  u.name = p.name  
}

In the C direction, only the first assignment is effective as an update, because u and its features are not writable (“object creation, modification, and deletion can only take place in the target model for the current execution”, Page 15 of [30]). Instead, it is treated as a condition which should be established by the relation. In the design direction, only the second assignment is effective as an update.

Finally, it is possible to define auxiliary query operations (RelationalTransformation :: helpers in Fig. 2), e.g.

query combineNames(cn : String, an : String) : String  
{ cn + "_" + an  
}

These operations can then be called in domains or in the when or where clauses of relations:

top relation Property2CMemberV2  
{ checkonly domain design p : Property  
  { owner = e : Entity { name = en },  
  name = n, isUnique = k }  
};  
  enforce domain C m : CMember  
  { memberOf = s : CStruct { },  
  name = combineNames(en,n), isKey = k }  
};  
  when { Class2CStruct(e,s) }  
}

2.5 Overall QVT-R transformation semantics

The logical semantics of a QVT-R transformation is that at termination, all the concrete top relations will be established between the source and target models—i.e. the target model will have been modified wrt the source model in order that all these relations hold. The execution semantics of a QVT-R transformation is that each execution semantics of a QVT-R transformation relation is applied to all source elements for which it is enabled, until the relation is established for all such elements. In addition, target elements which are not “required to exist” by the concrete top
relations should be removed. This is a somewhat ambiguous requirement [2], which we make precise in Sect. 5.

The QVT-R semantics can be characterised as being state based: only the states of the source and target models are relevant for relation application. In particular, no execution trace is persisted from one execution to another—however, an internal trace is available, whereby one relation can test if another has been established in the same execution of the transformation, via the when clause, as described above.

3 Issues in QVT-R semantics

Although QVT-R was devised with the intent of being a declarative language with a clear semantic interpretation [15], there have been a continuing series of problems over its semantics.

These problems can be grouped into the following main categories:

- Incompleteness and inconsistencies in the standard [31]. For example, issue QVT14-55 identifies incompleteness in the check-before-enforce mechanism, and issue QVT14-57 identifies gaps and problems in the RelToCore mapping in the standard, which (partially) defines the semantics of QVT-R via a translation to QVT-C.
- The state-based semantics and check-before-enforce mechanism are insufficient in some cases to support efficient or precise incremental updates of a target model in response to source model changes [37]. The mechanism can also have unintended effects in batch mode, e.g. performing \( n \) to 1 merging of elements which should not be identified.
- Different transformation problems require different criteria for rule application conditions and for target element matching/creation/update. The standard does not provide any capability for such flexibility, leading to contrived and complex specifications when a variant semantics is needed [7,36,37].

At the heart of these problems is the dichotomy between the aim that QVT-R should be a purely declarative language, defining the effect of a transformation independently of any algorithm/design, and the practical needs of specifiers to define modular, efficient and comprehensible specifications. Thus, the resolved issue QVT13-48 points out that intermediate states during a transformation execution must actually be considered.

The ATL language addresses the declarative/imperative dilemma by prohibiting read access to the target model. QVT-R uses when clauses with relation tests to provide read access to target elements—but there is no guarantee in QVT-R that the data of these accessed elements may not subsequently change in value and hence invalidate the relation that accessed the elements.

In UML-RSDS, we resolve the dilemma by expressing rules as logical predicates. The rules also have a procedural interpretation, and the sequential composition of these procedures establishes the logical conjunction of the rules interpreted as predicates (Sect. 4). Target data can be read in rules, but only at points where it has reached its final state. We will apply this same idea in our QVT-R semantics and impose conditions ((a) to (e) of Sect. 5.4) to enable QVT-R specifications to be interpreted in a declarative manner.

In contrast to the QVT standard, the implementation of QVT-R in the Medini QVT tool [11] is oriented towards the efficient execution of QVT-R for practical use. Medini QVT has become a de facto standard for QVT-R developers, as it has been the most widely used QVT-R tool for several years. The tool adopts a variant semantics (based on persistent traces), but also has limitations for incremental updates and a lack of flexibility in its semantics [38].

3.1 Incompleteness and inconsistencies in the QVT standard

The semantics of QVT-R is defined in different ways in [30]: Section 7.10 gives a semiformal description, supported by a more formal definition in Section B.2. Section 10 defines a mapping, RelToCore, from QVT-R to QVT-C. Each of these descriptions is incomplete, and RelToCore is inconsistent with Section B.2 [2]. It is not clear how the semantics operates for update-in-place execution (Issue QVT14-47).

Detailed criteria for transformation correctness and verification are omitted from [30]. For example, an update by a relation \( R \) may be inconsistent because a different or the same application of \( R \)—instead of an application of a different relation—has already assigned a conflicting value to a feature of a selected target element. As discussed in Sect. 2, relations with source domain constraints \( r = rx: R() \) on optional \( r \) references can fail to be applied to elements whose \( r \) value is empty/undefined (Issue QVT14-67). Multiple additions of elements to multiplicity features are not necessarily inconsistent (Issue QVT14-46), nor are multiple removals of elements from optional features, but mixtures of additions and removals, and other combinations of updates, can produce inconsistent behaviours.

As shown above, apparently correct rules can lead to subtle flaws, dependent upon the execution semantics of the rules. Three different mechanisms (check-before-enforce, keys and non-top relations) are available in standard QVT-R to support target element resolution and reduce non-determinism in relation execution, but these mechanisms also lead to semantic problems. Check-before-enforce is a coarse-grain mechanism, applying at the level of entire target domains. In situations where there are multiple target object templates
expressions, it would be preferable to have an alternative mechanism which tests individual target templates and only executes their actions if their predicate does not already hold for some target element (in which case such an element could be looked up and bound to the template root variable for use in subsequent target templates of the relation, or in the relation where clause).

The check-before-enforce mechanism is intended to support change propagation based only on the states of source and target models, but the lack of precise information about source–target correspondences may result in non-deterministic and imprecise change propagation. A change in the value of a source feature may result in deletion and re-creation of a target element, instead of a feature change in an application of the relation. However, the standard does not specify the relative order needed by the call are available [36].

Implicit deletion of elements (Section 7.10.2 of [30]) is unclear and ambiguous. It is expressed with respect to a single relation: that target elements \( t \) are deleted if they are not “required to exist” by a valid binding of source elements \( s \) in an application of the relation. However, \( t \) could also be “required to exist” by other relations, which would mean it should not be deleted. This implicit delete mechanism does not seem adequate to propagate removal of elements from collection-valued features. For example, if source class \( A \) and target class \( A1 \) have *-multiplicity features \( r \) and \( rr \):

```plaintext
top relation A2A1
{ enforce domain src a : A {}; 
enforce domain trg a1 : A1 {}; }

} top relation B2B1
{ enforce domain src b : B {}; 
enforce domain trg b1 : B1 {}; }

} top relation R
{ enforce domain src a : A
  { r = bx : B {}; }
  enforce domain trg a1 : A1
  { rr = b1x : B1 {}; }
  when { A2A1(a,a1) and B2B1(bx,b1x) } }
```

Removal of some \( bx \) from \( a.r \) does not necessarily lead to removal of any corresponding \( b1x \) (i.e. where \( B2B1(bx, b1x) \) holds) from \( a1.rr \), because \( b1x \) is still “required to exist” by \( B2B1 \). Only deletion of \( bx \) is propagated to deletion of \( b1x \) and its removal from \( a1.rr \).

The order in which different parts of a QVT-R relation are executed is not fully defined in [30]. The source domains and when clause are considered together, the source domains declare variables representing input model elements and features of these elements, and the source domains and when clause (possibly also involving target variables in relation tests) define constraints on these variables. For Property2CMember, \( p \) is bound to some Property instance, \( e \) is bound to \( p\_owner \), and \( n \) is bound to the value of \( p\_name \) and \( k \) to the value of \( p\_isUnique \). The when clause further binds or constrains variables which occur in it, e.g. in Property2CMember, the call `Class2CStruct(e, s)` restricts target variable \( s \) to be any `CStruct` already matched to \( e \) by `Class2CStruct`. On the target side of the relation, a binding of the remaining target variables is constructed (if possible) which satisfies the target domains and the where clause. In Property2CMember, this means finding/creating an instance \( m \) of `CMember` which satisfies \( m\_memberOf = s, m\_name = n \) and \( m\_isKey = k \). However, the standard does not specify the relative order of execution of the when clause and target domains, of different target domains. There is a requirement that all arguments of a where-call of a relation must be fully bound at the point of call; however, this does not ensure that all data needed by the call are available [36].

This lack of a definite execution order appears to have no benefit for efficiency or abstraction and complicates the definition of QVT-R specifications, particularly bx specifications [36]. Finally, the definition of rule inheritance (relation overriding) and of transformation extension is left unspecified in [30].

### 3.2 Issues with the RelToCore semantics

The RelToCore translation (Sect. 10 in [30]) defines an explicit semantics for QVT-R by translating QVT-R specifications into QVT-C. This translation is defined as a large and complex QVT-R transformation, which has many quality flaws [23]. The translation is also incomplete: various functions (getVarsOfExp) and rules (`RExpToMExp`, `TopLevelRelationToMappingForEnforcement`) are not given full definitions, and there is no treatment of collection templates (QVT-R issue QVT14-28) or relation overriding (issue QVT14-57). The translation of where-invoked relations requires a separate Core mapping for each pair of an invoker and invoked relation [7].

### 3.3 Issues with Medini QVT

Medini QVT [11] has been the most successful QVT-R tool to date; however, it differs from the QVT-R stan-
standard in that it does not support check-before-enforce and instead uses persistent traces to support change propagation. Apart from these differences, Medini QVT also has several omissions and errors: domain conditions are not supported, leading to additional complexity in bx specifications (cf. the bag2ast case of [36]); there is no transformation extension or relation overriding, leading to code duplication; sets cannot be passed as parameters to non-top relations (cf. the ecore2sql3 case of [36]). Container references are not writable, so that relations must be defined relative to container objects, resulting in multiple (> 2) levels of element structure in domains (cf. the ecore2sql case of [36]). This also complicates traceability, requiring the use of marker relations [6]. A number of OCL operators are not supported or are incorrectly supported. As in the standard, the execution order of where clauses is not determined by their textual order. However, the tool does enforce that top relations cannot be invoked in where clauses, which is still an open issue in the standard (issue 14-59). The Medini QVT change propagation strategy is fixed and cannot be varied. The strategy propagates attribute value changes, but movement of an element from one association end to another may result in target element deletion and creation.

The approach is inherently operational, and verification facilities and correctness criteria are missing, although Medini QVT provides debugging facilities.

4 UML-RSDS

UML-RSDS is a specification language based on a subset of UML class diagrams, use cases, activities and OCL 2.4. Applications, including transformations and subtransformations, are specified as use cases u which have preconditions Preu and a sequence of postcondition constraints Postu expressed in an OCL subset. Use cases can also have activities (behaviours) in a subset of UML activity language, expressed as pseudocode statements (“Appendix C”).

The UML-RSDS OCL subset excludes OclAny, and the invalid value, and uses classical logic instead of the 3-valued logic of the OCL standard. It uses the notation “&” for and, $\implies$ for implies, and $x : s$ as an alternative for $s \rightarrow \text{includes}(x)$\(^1\). The null value cannot be explicitly referred to but can be tested using the oclIsUndefined() operator.

4.1 UML-RSDS specification structure

A UML-RSDS specification consists of a class diagram together with one or more use cases, which may be linked by $\text{<<extend>>}$ or $\text{<<include>>}$ dependencies. For transformations, classes may be distinguished as belonging to the source or target metamodels. A use case is itself a UML classifier and may have local attributes and operations. It is also a behaviour (specification) and may have parameters, preconditions, postconditions, invariants and an activity. Postconditions provide a declarative specification of the use case behaviour, and are expressed as rules or constraints with the schematic form

$$E ::= \text{Pre} \implies \text{Post}$$

where $E$ is a source class and Pre and Post are boolean-valued expressions with context $E$, with Post possessing a procedural interpretation stat(Post) or statLC(Post) as a UML activity\(^2\). The context class $E$ is optional. As in UML operation postconditions, prestate forms $v@\text{pre}$ of variables and expressions $v$ can be used, to refer to the read-only state of $v$ at the start of the execution of the constraint on an $E$ instance.

For example, the Model2Program rule would be expressed as an OCL constraint:

```
UMLModel::

CProgram->exists( p | p.name = name )
```

In general, UML-RSDS rules are more concise and simpler in form than corresponding QVT-R rules.

Query operations may be defined locally to a use case, as in QVT-R transformations, and in addition, update operations can be defined, which may or may not return values. Query and update operations may also be defined for class diagram classes. All these types of operation can be called from UML-RSDS rules, but update operations should only be invoked in rule succedents.

Specifications can be internally composed sequentially by defining an activity of a composite use case uc which includes use cases uc1, \ldots, ucN. The activity of uc is defined as the sequential composition of the activities of the uc1.

4.2 UML-RSDS rule semantics

Constraints are logically interpreted as universally quantified first-order logic formulae, e.g.:

$$\forall m : \text{UMLModel} \cdot \exists p : \text{CProgram} \cdot p.name = m.name$$

The mathematical interpretation of UML-RSDS OCL is given in [21].

OCL predicates such as $p.name = name$ and $E \rightarrow \text{exists}(e | P)$ are also given an operational interpretation as UML

\(^1\) Using symbols for operators can help to make formulae more readable.

\(^2\) We say that such Post are effective for update.
activities, such as assignments and object lookup/creation actions (object creation of \( e : E \) implicitly updates the class extent collection \( E.allInstances() \) of \( E \) instances). For each predicate \( P \), an operational interpretation \( \text{stat}(P) \) defines an activity (written as a procedural statement) which attempts to establish \( P \) by reading a set \( rd(P) \) of features and class extents, and changing a set \( wr(P) \) of writable features and class extents \[18,21\]. This is the conventional behaviour.

In other words, execution of the succeedent is only attempted if it is not already true in the initial state of the rule execution. However, instead of using this global check over the postcondition, we can use the \( \text{stat}_{LC}(\text{Post}) \) activity, which uses local checks to avoid unnecessary updates. For example, \( \text{stat}_{LC}(x : y.r) \) for \(*\)-multiplicity \( r \) first tests if \( y.r \rightarrow \text{includes}(x) \) and only adds \( x \) to \( y.r \) if the test is false (for \( 0..1 \) multiplicity \( r \), only if \( y.r \) is empty).

In UML-RSDS, \textit{Class2CPointerType} could be written as:

\[
E ::
\begin{align*}
\text{Pre} \land \neg \text{Post}@\text{pre} & \implies \text{Post}
\end{align*}
\]

Using this mechanism, the equivalent of the Property 2CMember rule is:

\[
\text{Property::}
\begin{align*}
\text{CMember}->\exists( m | m.memberOf = \text{CPointerType}\{\text{owner.typeId}\}.\text{pointsTo} & \land \\
\text{m.name} = \text{name} \land m.\text{isKey} = \text{isUnique} )
\end{align*}
\]

where it is assumed that the \text{CPointerType} instance with \( \text{typeId} = \text{owner.typeId} \) already exists. In UML-RSDS, setting one end of a bidirectional association such as \( \text{members}/\text{memberOf} \) implicitly also updates the other end (here, adding \( m \) to the members of the selected \text{CStruct}). Likewise, removing an element from one association end implicitly updates the other end. Deletion of an element implicitly removes it from all association ends and class extents in which it resides. Deletion of an aggregation owner element deletes all the aggregation part elements (cascaded deletion). These implicit updates will also apply for the QVT-R semantics representation in UML-RSDS.

The lookup mechanism \( E[\text{val}] \) also applies to collections of key values: \textit{CType[valx]} for collection \( \text{valx} \) is the collection of instances of \textit{CType} with \textit{TypeId} \( \text{value} \) in \( \text{vals} \).

An alternative way of writing the above Property to CMember constraint is:

\[
\text{Property::}
\begin{align*}
p = \text{CPointerType}\{\text{owner.typeId}\} \implies \text{CMember}->\exists( m | \\
\text{m.memberOf} = p.\text{pointsTo} & \land m.\text{name} = \text{name} \land m.\text{isKey} = \text{isUnique} )
\end{align*}
\]

Here, \( p \) is an auxiliary variable, similar to an OCL \textit{let} variable or QVT-R local variable. It is implicitly \( \forall \)-quantified over the entire constraint.

There are two versions of the general existential quantifier. \( \rightarrow \exists \text{is}_{LC} \) is used to provide local “least-change” semantics in a conventional execution semantics context.

1. \( E \rightarrow \exists x(P) \) — in the case that \( E \) has a \textit{key} : \text{String} identity attribute, the \( \text{stat or stat}_{LC} \) design of this quantifier expression uses any key specification \( x.\text{key} = \text{value} \) present in \( P \) to lookup \( x \) (or create \( x \) and set its key if there is no existing \( x \) with the key value) and then attempts to establish the remainder \( Q \) of \( P \) for \( x \) using \( \text{stat}(Q) \) or \( \text{stat}_{LC}(Q) \). If \( E \) has no identity attribute \textit{key}, then with \text{stat} semantics a new \( x \) is always created and \( P \) is established for this \( x \) using \( \text{stat}(P) \).

2. \( E \rightarrow \exists x\text{is}_{LC}(P) \) — in the case that \( E \) has a \textit{key} : \text{String} identity attribute, the \( \text{stat or stat}_{LC} \) design of this quantifier expression uses any key specification \( x.\text{key} = \text{value} \) present in \( P \) to lookup \( x \) (or create \( x \) and set its key if there is no existing \( x \) with the key value) and then attempts to establish the remainder \( Q \) of \( P \) for \( x \) using \( \text{stat}_{LC}(Q) \). Otherwise, successive conjuncts of \( P \)
are considered, with the aim to find an instance \( x : E \) which satisfies as many of these conjuncts as possible. An attempt is then made to establish the remaining conjuncts for such an \( x \) using \( \text{stat}_{LC} \). A new instance \( x : E \) is created if the first (or only) conjunct of \( P \) cannot be satisfied by any \( E \) instance.

Used as queries, \( \exists \) and \( \exists_{LC} \) are equivalent, and test that \( E \rightarrow \text{select}(x | P) \) is non-empty.

### 4.3 UML-RSDS transformation semantics

UML-RSDS transformation constraints are unidirectional as rules, e.g. reading \( \text{UMLModel} \) instances and creating/updating \( \text{CProgram} \) instances. However, in some cases the rules of a transformation \( \tau \) can be syntactically inverted to operate in the reverse direction, e.g. the inverse of the \( \text{UMLModel} \) rule is:

\[
\text{CProgram::}
\quad \text{UMLModel} \rightarrow \exists u \left( u . \text{name} = \text{name} \right)
\]

This syntactic inversion is based on the semantic concept of a rule invariant. The conjunction of the inverted rules forms a transformation invariant \( \text{Inv}_{\tau} \) of \( \tau \) [25].

Unlike QVT-R, there is no rule inheritance mechanism in UML-RSDS. Rules are explicitly ordered in the transformation postcondition and are executed in this order on all applicable source elements. There is also no concept of designated read-only source models in UML-RSDS rules: individual rules may read and write any source or target data. However, to ensure logical consistency within a use case \( \tau \), the postconditions are explicitly ordered as \( r_1, \ldots, r_n \) in such a way that a rule \( r_i \) which writes data read by a rule \( r_j \) must precede \( r_j : i < j \). For example, the rule for \( \text{Entity} \) above must precede the rule for \( \text{Property} \), because the \( \text{Entity} \) rule writes \( \text{CPointerType} \), which is read by the \( \text{Property} \) rule. This condition is termed syntactic non-interference. The activity synthesised for such a \( \tau \) by design synthesis is \( \text{stat}(r_1); \ldots; \text{stat}(r_n) \).

The suffix \( \pre \) can be added to entity type names or feature names at points where they are read, to distinguish the read access from updates to these elements. \( v \pre \) is termed a pre-state expression. This enables a bounded-loop implementation of a constraint \( E :: P \) to be used, rather than a fixed-point implementation (which would repeatedly apply \( \text{stat}(P) \) to all instances of \( E \) until \( P \) becomes true for all instances).

Unlike QVT-R, UML-RSDS has an operator to explicitly delete model elements:

\[
x \rightarrow \text{isDeleted}()
\]

removes \( x \) from the model. Cascaded deletions and other implicit effects also take place, as described above. The operator can also be applied to sets of instances. This mechanism is often more convenient for specifying transformations involving element deletion, compared to the QVT-R technique of “deletion by selective copying” (copying the elements that are not to be deleted) [12].

For each form of statement \( S \), there is a definition of its weakest precondition with respect to some predicate \( P : [S]P \). This is the most general condition under which every execution of \( S \) establishes \( P \) [17,21]. If the \( \text{Post}_\tau \) constraints of a use case \( \tau \) are ordered so that no constraint \( r_i \) can be invalidated later by rules \( r_j \):

\[
r_i \implies [\text{stat}(r_j)]r_i
\]

for \( i < j \), then the semantic effect of the UML-RSDS use case \( \tau \) for batch-mode execution establishes the logical conjunction \( r_1 \& \ldots \& r_n \) of the use case postcondition constraints [18,21]. This is a direct semantics compared to the complexities of QVT-R semantics and facilitates verification using classical logic theorem provers. Transformation invariants \( \text{Inv}_{\tau} \) are particularly useful in reasoning about transformation correctness, using proof by induction over transformation steps [21].

Identity attributes provide a persistent trace mechanism, which enables the definition of incremental-mode execution of UML-RSDS transformations, as an extension of the standard batch-mode execution [25].

Table 1 summarises the differences between UML-RSDS and QVT-R. The facilities missing from QVT-R and present in UML-RSDS could potentially be added to QVT-R and supported by semantic translation. Facilities present in QVT-R and missing in UML-RSDS need to be translated into suitable representations in UML-RSDS.

### 5 Translation from QVT-R to UML-RSDS

In this section, we present the rationale for our approach to the semantics of QVT-R (Sect. 5.1) and define a detailed translational semantics for QVT-R separate-models transformations using UML-RSDS (Sect. 5.2). Section 5.3 illustrates the semantics on an example. In Sect. 5.4, we discuss properties of the semantics, in Sect. 5.5 consider how it supports incremental model changes, and in Sect. 5.6 consider variations and extensions of the semantics.

The general principle of the translation from QVT-R to UML-RSDS is to represent QVT-R top relations as UML-RSDS rules (OCL constraints), non-top relations as update operations, and queries as query operations (Table 2).
Table 1 Differences between QVT-R and UML-RSDS

| QVT-R facility                                      | UML-RSDS facility                                      |
|-----------------------------------------------------|-------------------------------------------------------|
| Check-before-enforce                                | Local check-before-enforce using \( \text{stat}_{LC} \) |
| Single and compound keys for elements                | Single keys                                            |
| Query operations                                    | Query and update operations, including cached query operations |
| \( \text{when} \) clause (lookup of individual target elements) | Element lookup by key, including lookup of multiple target elements |
| Unspecified rule execution ordering                 | Explicit rule execution ordering                       |
| Tool-dependent                                      | Internal transformation decomposition                  |
| transformation composition                          | via use case activities                                 |
| Rule inheritance and rule invocation                | No rule–rule relations                                 |
| Transformation extension                            | Transformation import                                  |

Table 2 Mapping from QVT-R to UML-RSDS

| QVT-R language element           | UML-RSDS language element            |
|----------------------------------|--------------------------------------|
| Relational transformation        | Use case representing the transformation |
| Top relation                     | Use case postcondition constraint    |
| Non-top relation                 | Update operation of use case         |
| Local variables                  | Auxiliary variables of constraint/operation |
| Query                            | Query operation of use case          |
| Key                              | Identity attribute                   |
| Domain                           | Property representing root variable, |
|                                 | Properties for other domain variables, |
|                                 | expressions for template expression |
|                                 | assignments and where condition      |
| \( \text{when} \) clause         | Expressions including trace tests    |
| \( \text{where} \) clause        | Expressions including update operation invocations |

5.1 Rationale for the semantics

As identified in Sect. 3, there are significant problems with the semantic basis of the QVT-R standard, particularly due to the check-before-enforce mechanism, state-based semantics, and the lack of variability and verifiability.

We carried out an analysis of 27 published QVT-R specifications, to identify how the main language features are used in practice. We took tutorial examples from the Medini QVT site projects.ikv.de/qvt and from the QVT-D project repository of examples originating mainly from ModelMorf: git.eclipse.org/c/mmt/org.eclipse.qvtd.git. We also took cases from Github repositories and from published papers [12,28,36–38]. Table 3 lists cases with their size in LOC, together with the kind of element mapping which is performed by the transformation, and any design patterns [20,25] or specification approach adopted. Table 4 gives a summary of the case approaches.

It can be seen that the most common strategy for mapping source to target elements is to enforce a 1–1 correspondence using QVT-R keys. Another common approach is to map individual source elements to a group of target elements in different classes: this vertical entity splitting is typical of refinement cases such as the ecore2sql versions [36]. Otherwise, update-in-place cases usually involve modifications to elements in-place, sometimes with element creations and deletions. Keys can also be used to merge multiple source elements with the same key value into a single target element—this is used in abstraction cases such as ast2dag [36]. The check-before-enforce mechanism is less frequently used to achieve such merging.

Overall, we conclude that key-based element matching and merging must be supported by any proposed QVT-R semantics and implementation. In the absence of key values, different approaches for target element resolution may be needed, such as 1–1 mapping in the Families2Persons case [37], or \( n - 1 \) merging of duplicate source objects (the HSM to NHSM cases). Thus, whilst check-before-enforce should be supported by a QVT-R semantics/implementation, it should not be mandatory. The use of persistent traces should be available as an alternative change propagation mechanism, because this enables more precise change propagation than check-before-enforce, but improved capabilities for using persistent traces should be provided, compared to the support in Medini QVT.

For separate-models transformations, we will consider four possibilities for a target element resolution strategy, to identify target elements \( t \) which establish required constraints \( P(s_v, t_v) \) on source and target variables \( s_v, t_v \), with respect to a binding \( s_v \mapsto s \) of source elements \( s \) to \( s_v \):

**Key based:** Use a key property \( k \) to locate target elements \( t \) with \( t.k \) value as required by \( P \). Update other features of \( t \) as required to establish \( P(s_v, t_v) \) with the binding \( t_v \mapsto t \).
Table 3  Element mappings and pattern usage in QVT-R cases

| Transformation                  | Size | Element mappings                  | Patterns/techniques                        |
|---------------------------------|------|-----------------------------------|--------------------------------------------|
| AbstractToConcrete              | 47   | Update-in-place                    |                                            |
| hsm2hsm (recursion)             | 48   | $n - 1$, 1–1 using keys           | Flattening                                  |
| ClassModelToClassModel          | 85   | 1–1 using keys                    | Recursive descent,                         |
|                                 |      |                                   | Map objects before links                   |
| HSM2FlatSM                      | 85   | $n - 1$ using custom strategy     | Flattening                                  |
|                                 |      |                                   |                                            |
| UmlToRel                        | 98   | 1–1 using keys                    |                                            |
| SeqToStm                        | 104  | 1–1 using keys                    |                                            |
| pn2pnw                          | 115  | 1–1 using keys                    | (vertical) Entity splitting/merging,       |
|                                 |      |                                   | Map objects before links                   |
| set2oset                        | 121  | 1–1 using keys                    |                                            |
| SeqToStmc                       | 149  | $1 - n$                           | (vertical) Entity splitting                |
| bag12bag2/bag22bag1             | 157  | $n - 1$ using keys                |                                            |
| ER2WebML/WebML2ER               | 190  | 1–1 using keys                    | Map objects before links                   |
| Ecore2copyQVT                   | 193  | $1 - n$                           | (vertical) Entity splitting                |
| cdrat                           | 202  | update-in-place                    | Deletion by selective copy                 |
| UmlToRdbms                      | 238  | 1–1 using keys                    |                                            |
| DNF_bbox                        | 263  | update-in-place                    | Guard against duplicate applications       |
| gantt2cpm                       | 378  | $1 - n$                           | (horizontal, vertical) Entity splitting    |
| DNF                             | 396  | Update-in-place                    | Guard against duplicate applications       |
| families2personsconfig         | 435  | 1–1 using custom Strategy         | (horizontal) Entity splitting              |
|                                 |      |                                   |                                            |
| dag2ast                         | 439  | $1 - n$ using recursion           | Marker relation,                           |
| ast2dag                         | 439  | $n - 1$ using keys                | Map objects before links                   |
| f2p/p2f                         | 462  | 1–1 using procedural Coding       |                                            |
| Bpmn2UseCase                    | 532  | $1 - n$                           | (vertical) Entity splitting                |
| Communication2class             | 1029 | 1–1 or $n - 1$                    | Explicit checks to avoid duplicate elements|
|                                 |      | Using keys                        |                                            |
| ecore2sql1,                     | 1120 | 1–1 using keys, $1 - n$           | Guard against duplicate applications,      |
| ecore2sql2,                     | 956  |                                   | Marker relation,                           |
| ecore2sql3                      | 960  |                                   | (vertical) Entity splitting                |
| RelToCore                       | 2038 | 1–1 using keys, $1 - n$           | (vertical) Entity splitting                |

Table 4  Summary of element mappings in QVT-R cases

| Element mapping approach         | # cases | Percentage |
|----------------------------------|---------|------------|
| 1–1 using keys                   | 14      | 52%        |
| 1–n vertical entity splitting     | 8       | 29%        |
| update-in-place                   | 4       | 15%        |
| $n - 1$ using keys               | 3       | 11%        |
| $n - 1$ check-before-enforce      | 2       | 7%         |
| $n - 1$ custom                    | 2       | 7%         |
| 1–n custom procedural             | 2       | 7%         |
| 1–n recursion                     | 1       | 4%         |

**Mandatory creation:** In the absence of target key properties, always create new target instances $t$ and update these to establish $P$ with the binding $tv \mapsto t$.

**Check-before-enforce:** In the absence of target key properties, search for elements $t$ which already satisfy all constraints of $P$ wrt the binding $sv \mapsto s$, and bind $t$ to $tv$. If such $t$ cannot be found, create new elements as for the previous case.

**Least-change check-before-enforce:** In the absence of target key properties, find target elements $t$ which are maximal partial or total matches for the constraints of $P$.  

 Springer
(and $t$ satisfies at least one constraint of $P$), and update the $t$ if necessary so that they satisfy $P$. Again, if no such $t$ can be found, create new target elements and update as required.

As regards efficiency, key-based and mandatory creation are the least costly strategies, in principle, whilst the other strategies require inspection of all target elements of particular classes. As regards correctness, both key based and least-change can result in direct conflicts between different rules: $t.f = v$ could be set for reused element $t$ and feature $f$ in order to satisfy $P$, conflicting with a previously set value $t.f = w$ established to satisfy another constraint $Q$. However, both mandatory creation and check-before-enforce can also result in conflicts, in cases where a 1-multiplicity reference $r$ has been set so that $t.r = rx$ and $w : rx.f$ to satisfy a requirement $w : t.r.f$ of $Q$, but $v : t.r.f$ is required by $P$, for $v \neq w$. The creation of a new $rx1$ and setting $t.r = rx1$, $v : rx1.f$ will invalidate $Q$. In these cases, $rx$ should be reused using least-change.

Our default semantics for target resolution is presented in Sect. 5.2, using key-based and mandatory creation. These are also the default mechanisms in Medini QVT and UML-RSDS (the stat interpretation of constraints, and $\rightarrow exists$ quantifier in UML-RSDS). Least-change can be defined using the statLC interpretation and the $\rightarrow existsLC$ operator of UML-RSDS, and check-before-enforce using a generalised let operator. These are presented in Sect. 5.6.

In order to ensure transformation correctness, we restrict transformations $\tau$ by five conditions (a) to (e):

- (a) “No secretly created objects”: Target data are write-only, and relations $R$ can only refer to model elements $e$ of source or target models which are explicitly declared in $R$ as object variables (as the rootVariable of a RelationDomain, or bindsTo of a TemplateExp).
- (b) “No inter-relation conflicts”: Different top relations do not have conflicting effects.
- (c) “No intra-relation conflicts”: No relation has internal conflicts in its effects (i.e. conflicts between different applications or within one application of the relation).
- (d) “No secretly deleted objects”: If $\tau$ refers to a target element $t$, then any target element $x$ from which deletion could propagate to $t$ must also be referenced by $\tau$.
- (e) “Call graph is surjective and non-cyclic”: There are no unused non-top relations and no cycles in relation calling dependencies.

(a) and (d) ensure that target elements created or updated by $\tau$ are always recorded in some trace tuple and cannot be deleted except via the trace-based semantics. (b) and (c) prevent internal semantic conflicts within $\tau$. (e) simplifies the semantic analysis. Although this prevents recursion between relations, recursion is permitted between query operations.

Formal definitions of these conditions are given in Sect. 5.4.

5.2 Translation for separate-models QVT-R transformations

A separate-models QVT-R transformation $\tau$ is semantically represented as a UML-RSDS use case $\tau'$. $\tau$ has rules which each have at least two domains, and these domains cannot all have the same typed model. The root variable names of distinct domains should be distinct. There should be at least one target domain and at least one source domain per relation. We assume that $\tau$ satisfies the semantic correctness properties of (a) to (e) of Sect. 5.4.

Key declarations key $E \{p\}$ of $\tau$ are interpreted as asserting that $p$ is an identity attribute of $E: E \rightarrow isUnique(p)$. In UML-RSDS, a single key can be used to look up elements, using the $\rightarrow exists$ quantifier as described in Sect. 4. In the case of two or more features forming a compound key, the translation is from key $E \{p_1, \ldots, p_n\}$ to the constraint

$$E \rightarrow for\{e_1\} \quad E \rightarrow for\{e_2\} e_1 = e_2, p_1 = e_2, p_1 & \ldots & e_1 \quad p_n = e_2, p_n \implies e_1 = e_2)$$

The query functions of $\tau.\text{helpers}$ are represented as query operations of the use case $\tau'$.

In our separate-models semantics, we give formal interpretations $Pres_m(m)$, $Con_m(m)$ and $Cleanup_m(m)$ as UML-RSDS transformation use cases for the update, creation and deletion phases of a relational transformation $\tau$ executed in the direction of typed model $m$. These transformations are invoked in the order $Pres_m(m); Con_m(m); Cleanup_m(m)$ from the UML-RSDS transformation $\tau'$ that represents the complete transformation $\tau$.

As in the QVT-R to QVT-Core mapping of [30], we use trace classes $R\text{trace}$ to record that relations $R$ on source domains $s$ (domains with model $m'$, $m' \neq m$) and target domains $t$ (with model $m$) have been successfully applied to particular model elements (Fig. 4). Traces enable relations to inspect the target model indirectly, without direct reference to target language classes or features. However, the information in traces therefore needs to be kept up-to-date with the actual source–target relationships: incremental changes to source or target models may invalidate existing source-target relationships, or result in the establishment of new relationships.

For each relation $R$, we define a trace class $R\text{trace}$, which has properties $x : E$ for each domain root variable $x : E$ of each domain $d$ of $R$ and for each object template root variable ($TemplateExp :: bindsTo in Fig. 2$) $x : E$ occurring in a domain $d$ of $R$ (rule 1 of the RelToCore mapping,
page 192 of [30]). Traces are independent of the execution direction of a transformation—this means that traces produced by an execution of the transformation in one direction can be used by a subsequent execution in a different direction. We refer to the root variable $x$ of a domain $d$, together with the object template root variables within the domain, as the object variables of the domain, $ovars_d$.

For relation $R$ executed in direction $m$, let $sdom$ be the sequence of source domains, i.e. the domains with model $m' \neq m$, including primitive domains. These are treated as checkonly domains for execution in direction $m$. $tdom$ is the sequence of target domains, domains with model $m$. These are considered to be enforce domains in execution direction $m$.

The source object variables $svars_R$ of $R$ are the object variables $\bigcup_{d \in sdom} ovars_d$ of its source (non-target) domains $sdom$, and the target object variables $tvars_R$ are the object variables $\bigcup_{d \in tdom} ovars_d$ of its target domains $tdom$. For separate-models transformations, $svars_R$ and $tvars_R$ are disjoint. The object variables of relation $R$ are denoted by $ovars_R$; these are $svars_R \cup tvars_R$. Thus, $R\text{trace}$ has properties corresponding to $ovars_R$. The set of all (free) variables declared in the domains of $R$ is denoted $avars_R$. This consists of the object variables and all other variables which occur as the referredProperty of PropertyTemplateItems in $R$, but does not include bound variables of quantifier or iterator expressions within $R$.

$whenvars_R$ is the set of variables occurring free in the when clause of $R$ ($Relation :: when.bindsTo$). These can include elements of $tvars_R$ used as parameters of relation calls (tests). $sourcevars_R$ is the collection of all $avars_R$ variables occurring in the source domains, and $svars_R$ is a subset of $sourcevars_R$ (the other variables of $sourcevars_R$ typically represent features of the $svars_R$ elements). $whenvars_R$ and $sourcevars_R$ are subsets of $avars_R$. The input variables $invars_R$ of a top relation $R$ are the object variables occurring in the source domains or the when clause. These are read and not written by $R$. For a non-top relation, all the domain root variables are also included in $invars_R$. The output variables of $R$ are the other object variables $ovars_R - invars_R$ of $R$; these are potentially written by $R$.

Table 5 summarises the notations used in the subsequent semantic definitions.

Table 6 summarises some of the key terminology used in the semantic definition.

Elements $elems$ of the source and target models are linked to one trace element $tr : R\text{trace}$ if $R$ has been successfully applied to the source elements of $elems$ to update or create the target elements of $elems$. These traces are tested when $R(a_1, ..., a_n)$ occurs as a rule call in a when clause (for either top or non-top relations $R$). The $a_i$ corresponds in order to the domain root variables $v_i$ ($R.domain.rootVariable$) of $R$, which are a subcollection of $ovars_R$.

A positive call $R(a_1, ..., a_n)$ in a when clause is logically interpreted as $tr : R\text{trace}$ & $a_1 = tr.v_1$ & \ldots & $a_n = tr.v_n$ where $tr$ is a new variable. If $a_i$ is a variable, the call binds the value of $tr.v_i$ to $a_i$. By using traces in this way we simulate logically this aspect of the Relations-to-Core semantics in [30]. A negative call $\text{not}(R(a_1, ..., a_n))$ is interpreted as $\text{not}(R\text{trace} \rightarrow \text{exists}(tr|tr.v_1 = a_1 \& \ldots \& tr.v_n = \ldots))$. 

---

Table 5 Variables associated with relations

| Notation   | Meaning                                                                 |
|------------|-------------------------------------------------------------------------|
| $sdom$     | The source domains $d \in R.domain$, including primitive domains        |
| $tdom$     | The target domains $d \in R.domain$                                     |
| $ovars_d$  | Object variables of domain $d$, including the root variable $s$         |
| $svars_R$  | $\bigcup_{d \in sdom} ovars_d$                                         |
| $tvars_R$  | $\bigcup_{d \in tdom} ovars_d$                                         |
| $ovars_R$  | $ovars_R$                                                                |
| $sourcevars_R$ | All variables bound in any $sdom$                                       |
|            | Includes $svars_R$                                                      |
| $targetvars_R$ | All variables bound in any $tdom$                                       |
|            | Includes $tvars_R$                                                      |
| $avars_R$  | All variables of any $sdom$ or $tdom$                                   |
| $sourcevars_R \subseteq avars_R$ | Variables free in when clause of $R$                                    |
| $whenvars_R$ | Input elements of $R$. For top relations: $svars_R \cup (whenvars_R \cap ovars_R)$ |
|            | For non-top relations: $svars_R \cup (whenvars_R \cap ovars_R) \cup$ $R.domain.rootVariable$ |
| $outvars_R$ | Output elements of $R$: $ovars_R - invars_R$                            |
|            | These are the object variables possibly instantiated by a successful application of $R$ |
Table 6  Semantic terminology

| Term                               | Meaning                                                                                                                                                                                                 |
|------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Target element resolution          | The process of inspecting the target model to identify if there are elements \( t \) which satisfy required constraints relative to source elements \( s \), and binding the \( t \) to target object variables \( o_{\text{vars}} \) if the \( t \) exist, or selecting or creating target elements to update and bind to target variables to establish the constraints |
| Elements referenced by a rule application of relation \( R \) | Source or target elements \( x \) bound to some \( v \in o_{\text{vars}} \) for a successful application of \( R \). Equivalently, \( x \) occurs in some \( R_{\text{strace}} \) tuple                                                                 |
| Elements referenced by a transformation execution | Elements referenced by any rule application in the transformation execution                                                                                                                                                                                                                         |
| Application condition of relation \( R \)  | Condition on \( o_{\text{vars}} \) determining if \( R \) can be applied to particular source model elements                                                                                                                                                                                   |
| Effective for update in direction \( m \)  | An expression \( e \) that has a defined \( \text{state}(e) \), and with \( \text{wr}(e) \) a non-empty subset of the variables \( \text{target}_{\text{vars}} \) from enforce domains with model \( m \)                                                                                          |

\( a_n \)). In this case, the \( a_i \) must already be bound prior to the call. However, negative calls cause semantic problems (Sect. 5.4).

The trace instances also provide a means to prevent re-application of a relation to the same source elements if it has already succeeded on them: relation \( R \) is only applied to source elements \( a_1, \ldots, a_k \) if there is not already a trace \( tr \in R_{\text{strace}} \) linked to these elements (the same approach is used in the translation of [8] from QVT-R to CPN). However, membership of \( R_{\text{strace}} \) is not necessarily equivalent to the validity of \( R \) on the linked elements, because of changes to source or target elements subsequent to the trace being created. Our semantics is designed to ensure that \( R_{\text{strace}} \)-linked elements do satisfy the logical interpretation of \( R \) at any points where they may be tested.

For the enforce semantics of a concrete top-level relation \( R \), we define three logical constraints in UML-RSDS:

- **Preservation constraints** \( \text{Pres}_R(m) \) expresses the effect of \( R \) when applied to tuples \( \text{elems} \) of source and target elements which are linked to one \( R_{\text{strace}} \) element, but which nonetheless may not satisfy the logical properties of \( R \)'s target domains—due to incremental changes of source or target model data. The effect of \( \text{Pres}_R(m) \) is to modify target model data linked to the trace, to re-establish the \( R \) target domain properties for \( \text{elems} \) relative to the source data. Traces which cannot be repaired are deleted.

  - **Construction constraints** \( (\text{Con}_R(m) \) expresses the effect of \( R \) when applied to source elements/tuples \( \text{selems} \) which are not in any \( R_{\text{strace}} \), and therefore have not yet had \( R \) applied to them. \( \text{Con}_R(m) \) establishes the logical properties of \( R \)'s target domains by looking up and updating target elements or by creating new target elements. It also creates a new trace instance for the \( \text{selems} \) and their matched target elements.

  - **Cleanup constraints** \( \text{Cleanup}_R(m) \) manages the deletion of target instances \( e : E \) that are not linked to any source instance via any valid trace \( R_{\text{strace}} \) for any relation \( R \) that may create \( E \) instances.

The constraints are grouped into three phases \( \text{Pres}_R(m) \), \( \text{Con}_R(m) \) and \( \text{Cleanup}_R(m) \) executed in this order. Within each phase, the constraints for individual top relations \( R \) are executed in the order they occur in the QVT-R specification. Within each relation execution, the source domains and \( \text{when} \) clause are tested together, and if these are satisfiable, then the target domains are executed, followed by the \( \text{where} \) clause. Execution order within target domains and the \( \text{where} \) clause follows textual ordering (left to right and top to bottom).

In the semantic representation, this is expressed by defining \( \text{Pres}_R(m) \) and \( \text{Con}_R(m) \) via an OCL formula \( \theta_R(m, \text{vars}) \) of the form

\[
\text{cpreds}(\text{sdm}, \text{vars}) \land \text{whenp} \implies \text{epreds}(\text{tdm}, \text{vars} \cup \text{whenvars} \cup \text{source-vars}) \land \text{wherep}
\]

which asserts that if the \( \text{when} \) clause and source domains are valid for the values of input variables \( \text{vars} \), then the effect of the target domains and \( \text{where} \) clauses is carried out, on \( \text{vars} \) together with the variables bound in the source domains and \( \text{when} \) clause. \( \theta_R(m, \text{vars}) \) is implicitly \( \forall \)-quantified over the variables of \( o_{\text{vars}} - \text{vars} \).

The LHS of \( \theta_R(m, \text{vars}) \) is the basis of determining the application conditions of \( R \), whilst the RHS can be adapted to give different semantics for target element resolution.

\( \text{whenp} \) is the logical interpretation of the \( \text{when} \) clause: the conjunction of the clause predicates, with relation calls treated as tests on the relation trace.

\( \text{wherep} \) is the logical interpretation of the \( \text{where} \) clause, the ordered conjunction of its predicate expressions, with non-top relation calls \( r(\text{vars}) \) treated as calls of the operation corresponding to \( r \), defined below. Only \( \text{where} \) clause predicates effective for update in the \( m \) direction are included in \( \text{wherep} \).

\( \text{cpreds}(\text{sdm}, \text{bound}) \) is the logical interpretation of the non-target domains \( \text{sdm} \), given a set of currently bound vari-
ables $\text{bound}$, and $\text{epreds}(\text{tdom, bound})$ is the interpretation of the target domains $\text{tdom}$, i.e. domains $d$ with model $m$. The scope of any $\exists$ or $\exists \text{LC}$ quantifiers introduced for target object variables in the $\text{epreds}$ formulae is extended over the remainder of the succedent, including $\text{where} ep$ (as in [30], Annex B). We define $\text{epreds}$ and $\text{epreds}$ in “Appendix B”.

We write $\theta_R(m)$ for $\theta_R(m, \text{ovars}_R)$. A predicate $\text{guard}_R(m)$ is formed as the antecedent $\varphi_R(m)$ of $\theta_R(m)$, with all variables apart from the $\text{ovars}_R$ variables $\exists$-quantified. That is, as an OCL formula it is

$$T_1 \to \exists(v_1) \ldots T_k \to \exists(v_k) \varphi_R(m) \ldots$$

where the $v_i$ are the variables in $\text{ovars}_R - \text{ovars}_R$. However, we eliminate $\exists$ quantifiers where possible by rewriting $T \to \exists(v[v = e \& P])$ to $P[e/v]$. For example, $\text{guard}_{\text{Model2Program}}(C)$ is $u : \text{UMLModel}$. The application condition $\text{guard}_R(m)$ of $R$ in direction $m$ is $\text{guard}_R(m)$ with target elements $\text{ovars}_R$ quantified out, so that it is a predicate on $\text{ovars}_R$ only.

The checkonly semantics of $R$ in the direction of $m$ asserts that at termination of $\tau$, for every tuple of $\text{ovars}_R$ elements that satisfy $\text{guard}_R(m)$, there is an extended tuple of elements for $\text{ovars}_R$ which satisfies $\theta_R(m)$ (Section 7.10.1 of [30]).

Regarding the $\text{overrides}$ clause in relations, we remove this by considering that all overridden relations are abstract and not executable—they are present in order to express commonalities between more specialised relations. Before applying the semantics, we merge the definition of an overridden relation into its concrete overriding relations (“Appendix D”). The overridden relation does not need a trace set. A $\text{when}$ test on an overridden abstract relation $R(\text{pars})$ is replaced by a disjunction $R_1(\text{pars})$ or ... $R_\alpha(\text{pars})$ testing the concrete relations $R_i$ that override $R$. This then becomes a test on their trace relations in the semantics. A $\text{where}$ call of a non-top abstract $R(\text{pars})$ is semantically expressed as a conditional

$$\begin{align*}
\text{if guard}_{R_1}[\text{pars/ovars}_{R_1}] & \text{ then } \text{Op}_{R_1}(\text{pars}) \\
\text{else} & \ldots \\
\text{else if guard}_{R_\alpha}[\text{pars/ovars}_{R_\alpha}] & \text{ then } \text{Op}_{R_\alpha}(\text{pars}) \\
\text{endif} & \ldots \text{ endif}
\end{align*}$$

The $\text{Op}_{R_i}$ are the operations representing the merged concrete relations $R_i$ overriding $R$. In a similar way, we can expand out a transformation composed using the $\text{extends}$ mechanism (“Appendix C”). Thus, in the following we will only consider concrete relations and transformations without $\text{extends}$.

5.2.1 Preservation constraints

The first execution phase $\text{Pres}_R(m)$ of $\tau'$ applies for incremental-mode execution only, using a persistent trace. For each relation $R$, $\text{Pres}_R(m)$ is an OCL constraint that defines $R$’s feature change propagation actions for elements that have already been matched (i.e. by a previous execution of $\tau$); it is defined schematically as

$$R\text{trace}@pre ::
\begin{align*}
\text{if guard}_R(m) & \text{ then } \theta_R(m) \\
\text{else } \text{self} & \to \text{isDeleted}() \\
\text{endif} & \ldots \text{ endif}
\end{align*}$$

If the guard holds, then the effect of $R$ is re-applied on the target elements linked to the trace element, to re-establish the logical property $\theta_R(m)$. Otherwise, the trace-linked elements cannot be updated to re-establish the property (because only target data can be updated), and the trace element is deleted.

Provided that target classes and features are not referred to in the $\text{when}$ clause or source domains, and that target data are only written and not read in $R$, and that $R$ itself is not invoked directly or indirectly in the $\text{when}$ or $\text{where}$ clause (the conditions (a) and (e) of Sect. 5.4), then $\text{Pres}_R(m)$ has disjoint read and write frames. In addition, elements of $R\text{trace}$ are only deleted, not created. This means that a polynomial-time complexity bounded-loop implementation of $\text{Pres}_R$ is sufficient, and we enforce this design choice by using the $@\text{pre}$ annotation on $R\text{trace}$.

The $\text{Pres}_R(m)$ constraints for both top and non-top $R$ form the postconditions of the use case $\text{Pres}_R(m)$. If the $\text{when}$ clause of $R$ calls $S$, then $\text{Pres}_S(m)$ must precede $\text{Pres}_R(m)$ in $\text{Pres}_R(m)$, because $\text{Pres}_S(m)$ writes $S\text{trace}$, which $\text{Pres}_R(m)$ reads. This ordering constraint is included in the correctness condition (e) of Sect. 5.4. That is, the $\text{Pres}_R(m)$ constraints are ordered in $\text{Pres}_R(m)$ according to the $\ll$ relation of Sect. 5.4.

$\text{Pres}_R(m)$ propagates element feature changes from source models to $m$. For example, $\text{Pres}_{\text{Model2Program}}(C)$ is

$$\text{Model2Program$\text{trace}@pre ::}
\begin{align*}
\text{if } u : \text{UMLModel} & \text{ then } (n = u\text{.name} \Rightarrow p\text{.name} = n) \\
\text{else } \text{self} & \to \text{isDeleted}() \\
\text{endif} & \ldots \text{ endif}
\end{align*}$$

This constraint propagates name changes from $\text{UMLModel}$ instances to $\text{CPrograms}$. In addition, if the $u$ element has been deleted (i.e. the trace reference $u$ is $null$), then the $\text{if}$ condition is $\text{false}$ and the trace is deleted.
5.2.2 Construction constraints

The second phase $Con_\tau(m)$ of $\tau'$ deals with the creation of new target elements and traces. It applies both for batch and incremental-mode execution. It consists of constraints $Con_R(m)$ for each top relation $R$. $Con_R(m)$ only applies if the source elements held in the $svars_R$ variables are not already linked to some $R\$trace instance:

\[
\begin{align*}
&: \text{cpreds}(sdom, \{\}) \land \text{whenp} \land \nonumber \\
&\text{not}(R\$trace@pre->exists(tr|&sv\in svars_R tr. sv = sv)) \implies \text{epreds}(tdom, \text{whenvars}_R \cup \text{sourcevars}_R) \land \text{wherepx}
\end{align*}
\]

$\text{wherepx}$ is $\text{wherep}$ right conjoined with the additional creation $R\$trace->exists(tr|&sv\in svars_R tr. sv = sv \land &h\in vars_R tr.tv = tv)$ of a new trace instance.

We use $R\$trace@pre if the antecedent because otherwise $R\$trace would be read and written by the constraint. However, a bounded-loop implementation is sufficient because the succedent can only reduce and not increase the collection of element tuples which satisfy the antecedent. Thus, as with $Pres_R$, we use the prestate expression to enforce a bounded-loop implementation. Any $\text{exists}$ or $\text{exists}_LC$ quantifiers introduced by $\text{epreds}(\ldots)$ in the succedent apply over all of the succedent following their introduction, by definition of $\text{epreds}$. Assuming the conditions (a), (e) as for $Pres_R(m)$, the write and read frames of $Con_R(m)$ are disjoint.

The $Con_R(m)$ constraints for concrete top relations $R$ are placed in $Con_\tau(m)$ in $<$ order as defined in Sect. 5.4. These constraints use the default target element resolution approach (key based with mandatory creation in the absence of keys) to satisfy $R$ in cases where the target elements do not already exist, and propagate element creation from source models to $m$.

For example, $Con_{\text{Model2Program}}(C)$ is:

\[
\begin{align*}
&: \text{u} : \text{UMLModel} \land n = u.\text{name} \land \\
&\text{not(Model2Program}$trace@pre->exists(tr | tr.u = u)) \implies \\
&\text{CProgram}$exists(p | p.name = n \land Model2Program$trace->exists(tr | tr.u = u \land tr.p = p))
\end{align*}
\]

5.2.3 Cleanup constraints

The third phase $Cleanup_\tau(m)$ of $\tau'$ consists of constraints $Cleanup_E(m)$ for each target entity $E$ of $m$. $Cleanup_E(m)$ is defined as:

\[
E :: \\
\text{not}(R1\$trace->exists(r1x|r1x.e1 = self)) \land ... \land \\
\text{not}(Rk\$trace->exists(rkx|rkx.em = self)) \implies \text{self->isDeleted()}
\]

where the $R_i$, $i = 1$ to $k$, are all the relations (top or non-top) in which the entity $E$ occurs as the type of some target object variable $e_i : E$ in $outvars_{R_i}$. All $Cleanup_E(m)$ constraints for entity types of $m$ are placed in a final $\tau'$ phase, $Cleanup_\tau(m)$. The constraint $Cleanup_E(m)$ reads and writes $E$, so it needs a fixed-point implementation: the constraint is re-applied until there are no $E$ elements satisfying its antecedent.

For example, if the UML2C transformation contains only one relation which can create $\text{CProgram}$ instances, $\text{Model2Program}$, then $Cleanup_{\text{CProgram}}(C)$ is:

\[
\text{CProgram} :: \\
\text{not(Model2Program}$trace->exists(tr | tr.p = self)) \implies \text{self->isDeleted()}
\]

The $Cleanup_\tau(m)$ constraints propagate element deletion from the source models to $m$, because if any element linked to a trace instance is deleted, so is the trace (the links from the trace to the elements are mandatory, as shown in Fig. 4).

A generally stronger version of the $Cleanup_E(m)$ constraint is:

\[
E :: \\
\text{not}(R1\$trace->exists(r1x|r1x.e1 = self \land r1x.gaurdR1(m))) \land ... \land \\
\text{not}(Rk\$trace->exists(rkx|rkx.em = self \land rkx.gaurdRk(m))) \implies \text{self->isDeleted()}
\]

$rx.P$ is $P$ with variables $u$ of $outvars_R$ in $P$ replaced by $rx.u$. This version of the cleanup constraint deletes any target element which does not occur in any $R_i$ trace that satisfies $\text{guard}_R(m)$. However, assuming the correctness conditions (a) to (e), all trace elements will satisfy their relation guard at termination of $Con_\tau(m)$, so the simpler version of $Cleanup_\tau(m)$ is sufficient in this case.

5.2.4 Non-top relations

Non-top relations $R$ are interpreted as operations with post-conditions defined in a similar manner to $Con_R(m)$. For a non-top-level relation $R$, a constraint $ConOp_R(m)$ is used to form the postcondition of an update operation $R_m$ which has as parameters the root variables of the (relation and primitive) domains of $R$. $ConOp_R(m)$ is defined as for $Con_R(m)$,
using \textit{cpreds} and \textit{epreds} but including the root variables of the relation domains in the bound variable sets:

\[
cpreds(sdom, \text{domain}.\text{rootVariable}) \land \text{whenp} \land \\
\neg(R\text{trace}@pre \rightarrow \exists (tr | \& \text{tvars}_R \land \text{dom} \land \text{rootVariable})) \\
= sv \land \& \neg\exists tv \land tr.tv = tv) \\
\rightarrow \\
epreds(tdom, \text{whenvars}_R \land \text{domain}.\text{rootVariable}) \\
\land \text{wherep}
\]

\textit{tvars}_R is \textit{tvars}_R \land \text{domain}.\text{rootVariable}. Target objects can be created using non-root object templates in target domains.

For example, the non-top relation \textit{NonTopProperty2CMember} has the interpretation in the \textit{C} direction as an update operation

\[
\text{NonTopProperty2CMember}_C(e : \text{Entity}, c : \text{CStruct})
\]

\textbf{post:}

\[
p : e.\text{ownedAttribute} \land n = p.\text{name} \\
\& k = p.\text{isUnique} \\
\neg(\text{NonTopProperty2CMember}_C@pre\rightarrow \\
\exists (tr | \\
tr.e = e \land tr.p = p \land tr.c = c)) \\
\rightarrow \\
\text{CMember}@exists( m | m : \\
c.\text{members} \land \\
m.\text{name} = n \land m.\text{isKey} = k \\
\neg(\text{NonTopProperty2CMember}_C@trace\rightarrow \\
exists (tr | \\
tr.e = e \land tr.p = p \land tr.c = c \land tr.m = m)) \\
\]

These operations are added as static owned operations of the use case \textit{Con}_R representing the transformation phase.

For \textit{Pres}_R, a different version of the operation \textit{R}_m is defined, with postcondition \textit{PresOp}_R(m):

\[
tr : R\text{trace} \land v_1 = tr.v_1 \land \ldots \land v_n = tr.v_n \implies \\
if \text{guard}_R(m) \\
then \theta_R(m) \\
else tr\rightarrow\text{isDeleted}()
\]

The \(v_1, \ldots, v_n\) are \textit{vars}_R. The operation \textit{R}_m is added as a static owned operation of \textit{Pres}_R.

This version of \textit{NonTopProperty2CMember}_C has the form:

\[
\text{NonTopProperty2CMember}_C(e : \text{Entity}, c : \text{CStruct})
\]

\textbf{post:}

\[
tr : \text{NonTopProperty2CMember}_C@trace \\
& e = tr.e \land p = tr.p \\
& c = tr.c \land m = tr.m \\
& if p : e.\text{ownedAttribute} \\
then (n = p.\text{name} \land k = p.\text{isUnique}) \implies \\
\]

\textbf{5.2.5 Overall transformation semantics}

For each execution direction of a separate-models transformation \(\tau\), the three phases described above are executed in the order \textit{Pres}_R(m); \textit{Con}_R(m); \textit{Cleanup}_E. The semantics generalises in a natural manner to consider execution directed at multiple target models, extending [30].

Figure 5 shows the data dependencies of \textit{Con}_R and \textit{Pres}_R wrt trace entities, for the default semantics. \textit{where} is the recursive closure of relation calling though \textit{where} clauses, likewise for \textit{when}. An arrow \(d \rightarrow R\) means that \(R\) can read data \(d\), and an arrow \(R \rightarrow d\) means that \(R\) can write \(d\).

This shows that in order to avoid cycles in data dependencies, the constraints within transformations \textit{Con}_R and \textit{Pres}_R, quite strict constraints must be placed on how relations depend on each other via \textit{when} and \textit{where} clauses. These conditions are encoded in condition \((e)\) of the following section.

\textit{Cleanup}_E reads and writes \(E\), and reads \textit{Rtrace} for any relation \(R\) with an \textit{outvars}_R variable of type \(E\). It may also write target classes \(F\) which are affected by deletion propagation from \(E\). To ensure that trace classes linked to \(F\) are not also written, we assert the condition \((d)\) that target elements referenced by \(\tau\) (occurring in some trace tuple) cannot be affected by cascaded deletion from target elements that are not referenced by \(\tau\) (not occurring in any trace tuple).

\textbf{5.3 Example of the semantics}

As an illustration of the semantics, we show the translation of a simple version of the Families to Persons case of [1] to UML-RSDS. Figure 6 shows the metamodels; the transformation consists of a single rule:
transformation tau(source: MM1, target: MM2)
{
  top relation FamilyMember2Male
  { checkonly domain source fm :
      FamilyMember
      { membername = n }
  ;
  enforce domain target m : Male
  { name = fm.memberOf.familyname + "", " + n }
  when
  { fm.memberOf.father->includes(fm)
    or
    fm.memberOf.sons->includes(fm)
  }
}

The semantic translation of this transformation is:

**** UML-RSDS of QVT-R is:
Use Case, name: tau$Pres
FamilyMember2Male$trace@pre::
  not(fm : FamilyMember & fm.memberOf.father->includes(fm)) &
  not(fm : FamilyMember & fm.memberOf.sons->includes(fm)) =>
  self->isDeleted()

FamilyMember2Male$trace::
  n = fm.membername & fm.memberOf.father->
  includes(fm) =>
  m.name = fm.memberOf.familyname + "", " + n

FamilyMember2Male$trace::
  n = fm.membername & fm.memberOf.sons->
  includes(fm) =>
  m.name = fm.memberOf.familyname + "", " + n

Use Case, name: tau$Con

Use Case, name: tau$Cleanup

The Pres constraints concern incremental execution, i.e. re-application of tau to a modified source model. The first Pres constraint removes a pair (fm, m) of family member fm and male m from the relation trace if fm no longer satisfies the conditions to be mapped to m. The second and third propagate name changes from fm and fm’s family to m’s name.

The Con constraints apply to initially map a families model to a persons model and to propagate the introduction of new source elements to new target elements. The antecedent of these constraints includes a check that fm is not already mapped to some m.

Finally, the Cleanup constraints remove target elements that are not related to some source element. Notice that the inverse direction of the trace-to-class association in Fig. 4 is used to optimise the Con and Cleanup constraints.

The translation can be used to show semantic equivalence of different versions of the transformation, to support bidirectionalisation of the transformation. Further details of this example are given in https://nms.kcl.ac.uk/kevin.lano/qvt2umlrsds.pdf.

5.4 Properties of the semantics

The three phases Pres(τ), Con(τ), Cleanup(τ) of τ’ should establish the following consistency relations Relτ of τ directed at m:

1. That any existing target instance tx : E of any entity type E of m must appear as a target property of some
trace: $E \rightarrow \forall (tx | R1$\text{trace}$\rightarrow \exists (r1.t1 = tx) \text{ or } ... \text{ or } Rk$\text{trace}$\rightarrow \exists (rk|rk.tk = tx))$ where the $Ri$ are as for Cleanup$_E(m)$.

2. That all instances $tr$ of $R$\text{trace}$ satisfy the logical relation $\text{guard}_R(m) \& \theta_R(m)$ defined by $R$, for each relation $R: tr.\text{guard}_R(m) \& tr.\theta_R(m)$.

3. That any tuple of source elements for $svars_R$ that satisfies $\text{guard}_R(m)$ is linked to some $R$\text{trace}$ instance $tr$, for top relations $R$.

Conditions (2) and (3) imply that for any source element tuple $x$ for $svars_R$ which satisfies $\text{guard}_R(m)$, for top relation $R$, there is an extended tuple $y$ of elements for $ovars_R$ such that $y$ is in $R$\text{trace}$ and satisfies $\text{guard}_R(m) \& \theta_R(m)$. This means that the checkonly semantics of $R$ is satisfied at termination of $\tau'$.

However, the following equivalence is not generally true (either at termination or during transformation execution) for a top relation $R$, for any tuple $vals$ of instantiating elements for the object variables $ovars_R = \{v_i\}$ of $R$: 

\[ R$\text{trace}$\rightarrow \exists (tr|&_i(vals_i = tr.v_i)) \equiv (\text{guard}_R(m)[vals/ovars_R] \& \theta_R(m)[vals/ovars_R]) \]

That is, membership of $vals$ in the trace is equivalent to successful execution of $R$ on $vals$. This fails in the $\Leftarrow$ direction because it is possible for a tuple $vals$ to satisfy $\text{guard}_R(m)(vals) \& \theta_R(m)(vals)$ but not be in $R$\text{trace}$, because $vals$ has not been processed by $R$. For example, consider the transformation with top rules:

\begin{verbatim}
 top relation R1
  { enforce domain src a : A { name = n }; 
    enforce domain trg b : B { name = n }; }

 top relation R2
  { enforce domain src c : C { value = v }; 
    enforce domain trg b : B { value = v }; }

 where source element $a1 : A$ and target elements $b1 : B$, $b2 : B$ all have the same name and source element $c1 : C$ has the same value as $b2$. One possible execution would match $a1$ to $b1$ and $c1$ to $b2$, so that at completion of the transformation 
\[ \text{guard}_{R1}(trg)[a1/a] \& \theta_{R1}(trg)[a1, b2/a, b] \] 
holds but not $(a1, b2) \in R1$\text{trace}$. $R1$ only selects one possible $B$ instance to match to $a1$, in this case $b1$.

The $\Rightarrow$ direction can also fail, because conflicts between relations may invalidate existing traces. For example, if $id : \text{String}$ is a key for target class $B$, and source classes $A, C$ have String-valued features $id, value$ and $name, value$ respectively, then the relations:

\begin{verbatim}
 top relation R1X
  { enforce domain src a : A  
    { id = ix, value = v }; }

 top relation R2X
  { enforce domain src c : C  
    { name = n, value = v }; 
  
  } top relation R1
  { enforce domain src a : A { name = n }; 
    enforce domain trg b : B { id = ix, value = v }; }

 top relation R2
  { enforce domain src c : C { value = v }; 
    enforce domain trg b : B { id = n, value = v }; }
\end{verbatim}
can conflict, with some $b : B$ created by $R1X$ and hence with $b = tr.b$ for some $tr : R1X$\text{trace}$, but also matched by key to some $c : C$ by $R2X$ and hence potentially modified so that $b.value$ no longer satisfies $\theta_{R1X}(trg)$. Another situation of conflict between relations is if a relation $R2$ has a negative $\text{when}$ test $\text{not}((R1(v))$ on another relation. Subsequent execution of $R1$ may invalidate the guard of $R2$ and hence invalidate the $\Rightarrow$ direction of the equivalence. Such tests should be replaced by alternative guards when using the “Guard against duplicate applications” pattern.

To address these issues, we formulate five correctness conditions for QVT-R specifications $\tau$ directed at a model $m$. These conditions ensure that the postconditions (1), (2), (3) are established by the semantic interpretation $\tau'$ of $\tau$:

- (a) “No secretly created objects”: No target features/entity names of $TL$ occur in the $\text{when}$ clause or source domain patterns of any relation, and relations can only refer to target features/class names via explicitly declared object variables $t : T$ and their features, in domain templates. Target data cannot be read, only written, in the parts of a relation that are effective for update in direction $m$;
- (b) “No inter-relation conflicts”: If two top relations both write (directly or via called relations) to the same target features or entity types, these updates are non-conflicting (i.e. they create/update disjoint sets of elements of the same target entity type, or update disjoint sets of features of the same instances of the entity). If all updates to a reference feature $f$ of *-multiplicity are additions, these are also non-conflicting. Similarly, if all updates to a 0..n or * multiplicity reference feature $f$ are all removals, these are non-conflicting;
- (c) “No intra-relation conflicts”: There are no self-conflicts or internal conflicts of a relation $R$, i.e. one application of $R$ cannot invalidate the same or a different application, e.g. by defining contradictory values for the same target data (Sect. 6). The $\text{where}$ clause should not conflict with target domain templates;
- (d) “No secretly deleted objects”: If target element $t : A$ is referenced by $\tau$, so also is any target element $b : B$ linked to $t$ via a mandatory target reference $A :: r$ (i.e. where $r$ is of 1 or 1..n multiplicity at the $B$ end), and
so is any aggregation owner element \( b : B \) linked to \( t \) by an aggregation association (Fig. 7). Another way to express this is if \( t \) appears in any trace tuple of some \( \tau \) relation, \( b \) must also appear in a trace tuple, of the same or a different relation;

- (e) “Call graph is surjective and non-cyclic”: Relation calls in \( \text{when} \) clauses must be positive (including disjunctions or conjunctions of positive calls). Any recursion in queries should be shown to be always terminating. There should be no recursion between relations via \( \text{when} \) or \( \text{where} \) clauses\(^3\). All non-top relations are called from at least one top relation directly or indirectly. No relation can be called via both \( \text{when} \) and \( \text{where} \) clauses starting from one relation. More precisely, it should be possible to order the top relations as \( R_1, \ldots, R_n \) such that:

1. \( R_i \) called in \( R_j, \text{when} \) \( \implies \) \( i < j \)
2. \( S \) called in \( R_j, \text{where*} \) and \( R_i \) called in \( S, \text{when*} \) \( \implies \) \( i < j \)
3. \( S \) called in \( R_i, \text{where*} \) and \( S \) called in \( R_j, \text{when*} \) \( \implies \) \( i < j \).

Condition (a) prevents relations from referring to arbitrary target elements or to the extents \( T.allInstances() \) of target classes \( T \). Thus, predicates such as \( v = T.allInstances() \rightarrow \text{size}() \) or \( \exists! \{t|P \} \) are prohibited. The condition ensures that all target elements whose data are read by or written to in a relation must also be linked to some trace of the relation. (a) and (e) ensure that the constraints in the \( \text{Pres}_\tau \) and \( \text{Con}_\tau \) phases can be ordered to avoid cycles in data-dependency.

These conditions ensure that cycles cannot arise in the trace data dependencies of Fig. 5. This means that bounded loop execution can be used for the \( \text{Con}_\tau \) and \( \text{Pres}_\tau \) phases, resulting in polynomial time complexity for both batch and incremental execution modes. (b) and (c) make precise the conditions for specification consistency based on clashes of bindings discussed in [30]. Condition (d) is a further consistency requirement which avoids conflict between the creation and deletion actions of relations. Condition (e) is the ‘no cycles’ condition also assumed by [8]. Recursion in specifications can hinder bidirectional execution and can usually be replaced by the Map Objects before Links pattern, \( \text{closure} \) expressions [28] or \( \forall \) quantifiers.

**Theorem 1** If the correctness conditions (a) to (e) hold for a separate-models transformation \( \tau \), then a partial execution order \( \triangleleft \) can be defined on all concrete relations based on the order \( R_i < R_j \) of top relations defined by \( i < j \). For top relations, \( \triangleleft \) is defined as \( \llcorner \). For non-top \( R \), top \( P \), \( R \triangleleft P \) means that any top \( S \) with \( R \in S, \text{where*} \) must have \( S < P \). For non-top \( P \), top \( R \), \( R \ll P \) means that any top \( S \) with \( P \in S, \text{where*} \) must have \( R < S \). For non-top \( P \), \( R \ll P \) means that any top \( S \), \( T \) with \( R \in S, \text{where*} \), \( P \in T, \text{where*} \) have \( S < T \).

**Proof** The transitivity property for \( \ll \) can be established by considering different cases of \( \ll \). Clearly \( P \ll R \) implies \( P \neq R \). If \( P \ll R \), then \( R \ll P \) is not possible:

- If \( P \) and \( R \) are both top relations, then \( P < R \) and hence \( \neg (R < P) \).
- If \( P \) is top, \( R \) non-top, then any top \( S \) with \( R \in S, \text{where*} \) has \( P < S \). But \( R \ll P \) would imply that \( S < P \).
- Likewise for non-top \( P \), top \( R \).
- If both \( P \), \( R \) are non-top, then any top \( S \), \( T \) with \( P \in S, \text{where*} \), \( R \in T, \text{where*} \) has \( S < T \). But \( R \ll P \) would imply \( T < S \).

Thus, provided that every non-top relation \( P \) has some top \( R \) with \( P \in R, \text{where*} \), the result follows. \( \Box \)

This suggests a way to organise QVT-R transformations: list the top relations in \( \triangleleft \) order, and group the non-top relations of \( R, \text{where*} \) for top relation \( R \) together with \( R \).

Note that it is possible for \( \tau \) to satisfy the correctness conditions (a) to (e) in one execution direction but not in others. For a \( \text{bx} \), we require that the conditions hold in all execution directions. This means that the transformation can be executed in multiple directions to synchronise models which may be modified independently. This capability fails for the standard specification of the UML to RDBMS QVT-R transformation [28]. A consequence of (a) is that \( \text{when} \) clauses of \( \text{bx} \) can only contain relation tests and equality/inequality tests on variables, and no occurrences of source or target features or class names.

**Theorem 2** For both batch-mode and incremental-mode execution of a separate-models transformation \( \tau \), the correctness conditions (a) to (e) ensure that \( \tau \) establishes or preserves the properties (1), (2) and (3).
Proof For batch-mode execution, assuming (a), (b), (c), (d), (e), the \(Con_{t}(m)\) phase establishes invariant (3), since if tuple \(v\) satisfies \(guard_R(m)\) for concrete top relation \(R\) at termination of \(\tau\), i.e. \(guard_R(m)[v/svars_R]\), it also did so at the start of the \(Con_{t}(m)\) execution step of \(Con_{t}(m)\)—because \(guard_R(m)\) depends only on source model and trace data which are read-only subsequent to the start of \(Con_{R}(m)\) (if \(R\) reads \(S\$trace\) via a relation test of \(S\) in a when clause, then \(S \ll R\) in the execution ordering).

By definition of \(guard_R\), there is an extended tuple \(w\) of \(v\) which includes \(tvars_R\) elements in \(whenvars_R\), which satisfies \(guard_R(m)\), i.e. \(guard_R(m)[w/invars_R]\). If \(w\) is not already linked to an \(R\$trace\) element, then \(Con_{R}(m)\) is applied to \(w\), and \(w\) extended to \(u\) with created/matched target elements of \(outvars_R\) then satisfies \(guard_R(m)\) and \(\theta_R(m)\), and is linked to a new \(R\$trace\) element. Trace elements can only arise in this manner, and are not removed by \(Cleanup_{t}(m)\) (because of condition (d)); hence, (2) and (3) hold. The validity of \(guard_R(m)\) and \(\theta_R(m)\) for \(u\) in \(R\$trace\) is not affected by subsequent \(Con_{t}\) applications in \(Con_{t}(m)\), either of \(R\) or another rule, by conditions (b), (c), (e). Condition (e) also ensures termination of the \(Con_{t}\) phase.

The \(Cleanup_{t}(m)\) actions do not invalidate \(guard_R(m)\) or \(\theta_R(m)\) for \(u\) in \(R\$trace\) because they cannot modify elements that are in a trace. In particular, (a) means that \(R\) only created/modified target elements that (after its execution) occur in \(R\$trace\), and hence cannot be affected by \(Cleanup_{t}\). For separate-models transformations, the \(Cleanup_{t}(m)\) constraints establish (1) and do not invalidate (2) because they only update the target model, not the source or traces. Condition (d) ensures that \(Cleanup\)-initiated cascaded delete does not delete elements that are in traces. In situations where a target \(a : A\) element would be deleted if a target \(b : B\) element is deleted, if \(a\) is in some trace tuple, so must \(b\) be.

For incremental-mode execution, if conditions (1), (2), (3) already hold, then none of the phases \(Pres_{t}(m)\), \(Con_{t}(m)\) or \(Cleanup_{t}(m)\) have any effect. (2) means that the \(Pres_{t}(m)\) constraints have no effect. (3) means that no \(Con_{R}(m)\) constraint is enabled. (1) means that no elements satisfy the criteria to be deleted by \(Cleanup_{t}(m)\).

Assuming the above conditions (a) to (e) for transformation \(\tau\), \(Pres_{t}(m)\) establishes (2) for any incremental change in the source model, since for each top relation \(R\), if a trace element no longer satisfies \(guard_R(m)\), then it is deleted; otherwise, its linked elements are updated if necessary to re-establish \(guard_R(m)\) & \(\theta_R(m)\). Deletion of source elements will delete any trace that they belong to. \(Con_{t}(m)\) does not invalidate any existing traces, because of the non-interference and ordering properties (b), (c) and (e), but may create new traces for new source elements, or for existing source elements which now satisfy some relation guard. Likewise, \(Cleanup_{t}(m)\) does not invalidate existing traces, but deletes target elements which no longer appear in traces. Thus, (1) and (3) are established.

Changing the definition of target element resolution does not affect this proof; however, the definition affects conditions (b) and (c) of rule consistency. In some execution scenarios, only one subtransformation needs to execute in order to enforce (1), (2) and (3). In batch mode with an initially empty target model and empty trace, only the \(Con_{t}(m)\) use case is necessary.

Corollary If conditions (a) to (e) hold for \(\tau\) in direction \(m\), then \(\tau\) satisfies the checkonly semantics of [30] in the direction \(m\).

Proof Theorem 2 shows that the conditions (1), (2), (3) are established by \(\tau\) at its termination. For each concrete top relation \(R\) of \(\tau\), conditions (2) and (3) imply that for any source element tuple \(x\) for \(svars_R\) which satisfies \(guard_R(m)\), there is an extended tuple \(y\) of elements for \(outvars_R\) such that \(y\) is in \(R\$trace\) and satisfies \(guard_R(m)\) & \(\theta_R(m)\).

Theorem 3 The presented semantics is consistent with the Relations-to-Core semantics of [30].

Proof We consider the 6 principal rules of Sect. 10 of [30].

- Rule 1: The construction of \(R\$trace\) in our semantics is identical to that of \(TR\) in [30].
- Rule 2: (2.1): the relation \(R\) when clause is represented in the \(guard_R\) predicate. (2.2): relation domains are either expressed in the source \(cpreds\) condition or target \(epreds\) constraint. (2.3): an instance of \(R\$trace\) is tested (in \(Pres_R\) or created (in \(Con_R\)). (2.4): each source property template item becomes an equation/membership test in \(cpreds\); each target property template item becomes an assignment/element addition in \(epreds\). (2.5): when predicates become tests on the LHS of \(Pres_R\) and \(Con_R\). (2.6): predicates of the where clause become RHS predicates of \(Pres_R\) and \(Con_R\). In contrast to [30], we also express relation calls in the where clause as predicates (calls of operations).
- Rule 3: We only consider relations with at least one enforceable domain.
- Rule 4: (4.1, 4.2): We permit multiple enforced target domains. (4.3): Each such domain has a realised root variable (quantified by an \(exists\) or \(exists\) \(LC\) operator) in \(epreds\), if it is not also an input to \(R\). Property template items in enforced target domains become assignments/element additions in \(epreds\). (4.4): where clause predicates are included in \(where\) \(px\), which is within the scope of the \(epreds\) realised variable quantifiers. (4.5): source domains are mapped as in Rule 2. (4.6): we do not model black-box operations, but these could be formalised as updater operations in UML-RSDS.
• Rule 5: (5.1): we represent an invoked relation \( S \) by an updator operation \( OpS \), called from the constraints representing the relations that call \( S \). (5.2, 5.3): The root variables of the \( S \) domains become parameters of \( OpS \) and are instantiated in the call.

• Rule 6: As for rule 5, we treat relation calls as conventional procedure calls.

Our semantics also gives complete definitions for incomplete parts of the Relations-to-Core mapping: the \( \text{getVarsOfExp} \) function, and the \( \text{RExpToMExp} \) and \( \text{TopLevelRelationToMappingForEnforcement} \) rules.

5.5 Incremental execution and change propagation

Source-to-target change propagation of the following source model changes is supported for separate-models transformations in incremental mode by our semantics:

• Addition of a new source element \( sx : ST \)—if this element satisfies some \( \text{guard}_R(m) \) of a relation \( R \) with source domain \( s : \text{ST} \) for \( \text{ST} \) equal to \( ST \) or a supertype of \( ST \), then \( sx \) will be processed by \( \text{Con}_R(m) \) and appropriate target and trace elements selected or created. Further top relations may consequently also become enabled and will be applied.

• Changes to a 1-multiplicity reference \( f \) or non-key attribute \( att \) of an existing \( s : \text{ST} \)—if \( s \) occurs in some trace \( \text{R$trace} \) and \( \text{guard}_R(m) \) remains true for \( s \), then \( \text{Pres}_R(m) \) applies to update linked target elements appropriately. If \( s \) is in some \( \text{tr} : \text{R$trace} \) but \( \text{guard}_R(m) \) is falsified by the change of \( \text{flattn} \) value, then \( \text{tr} \) is removed from \( \text{R$trace} \) by \( \text{Pres}_R(m) \), and \( \text{Cleanup}_R(m) \) removes any target elements that depended exclusively on \( s \). If \( s \) does not occur in any trace, but the change in \( \text{flattn} \) now validates some \( \text{guard}_R \), then \( \text{Con}_R(m) \) is applied to \( s \).

• Addition of an element to a collection-valued feature \( f \) of existing element \( s : \text{ST} \)—if \( s \) is in \( \text{tr} : \text{R$trace} \) and \( \text{guard}_R(m) \) remains true, then \( \text{Pres}_R(m) \) propagates the change to corresponding target elements. The cases of \( \text{guard}_R(m) \) being falsified and \( s \) not in \( \text{R$trace} \) are handled as for the previous case.

In general, we use conditions on traces to selectively enable specific constraints for particular incremental changes, instead of re-applying the entire transformation. This is based on a static scheme, rather than a dynamic mechanism as in [10], but the performance was found to be satisfactory on all the test cases of Sect. 8. A more elaborate static scheme for managing incremental changes by additional operations is defined by [33]; however, this considerably enlarges and complicates the semantic representation of a transformation, so making it harder to understand and analyse.

5.6 Variant semantics and extensions for QVT-R

The above semantics can be adapted to the cases of check-before-enforce and least-change semantics for target element resolution. We also consider the case of non-persistent traces and propagation of element removal. In addition, a semantics can be given to internal transformation composition.

5.6.1 Check-before-enforce semantics

Modifying the semantics of a relation \( R \) to use check-before-enforce target resolution means changing the definition of the succedent of \( \text{Con}_R \) (or \( \text{ConOp}_R \) for non-top \( R \)). Two cases need to be distinguished:

1. Either there is already a tuple of elements of \( \text{outvars}_R \) which satisfy the required constraints

\[
\text{epreds}(\text{tdom}, \text{whenvars}_R \cup \text{sourcevars}_R \cup \text{outvars}_R) \land \text{wherep}
\]

In this case, the elements are selected and a new \( \text{R$trace} \) instance created for \( \text{ovars}_R \), or

2. No such tuple exists, and new target elements need to be created and updated using \( \text{epreds}(\text{tdom}, \text{whenvars}_R \cup \text{sourcevars}_R) \land \text{wherep} \), and the resulting \( \text{ovars}_R \) elements linked in a new trace element.

In order to achieve this logic, we need an extended OCL \( \text{let} \) operator, which generalises the version defined in [29].

This has the form

\[
\text{let vars be P in Q}
\]

where \( \text{vars} \) are a list of variables, \( P \) are constraints on the \( \text{vars} \), and \( Q \) is a constraint which has a \( \text{stat}(Q) \) interpretation and does not write to the variables of \( P \). This generalised \( \text{let} \) has a procedural interpretation as an \( \text{any} \) statement [17] and hence has a well-defined predicate transformer semantics.
The check-before-enforce version of $Con_R(m)$ is therefore formalised as:

\[
\begin{align*}
\tau &::= \text{epreds}(\text{tdom}, \{\}) \& \text{whenp} \& \\
&\quad \text{not}(R\text{Strace}\@\text{pre}\rightarrow\text{exists}(\text{tr}|\&_{v \in \text{outvars}_{R\text{tr}}}\text{tr}.

sv = sv)) \implies \\
&\quad \text{if } 3v : (\text{epreds}(\text{tdom}, \text{whenvars}_{R} \cup \\
&\quad \text{sourcevars}_{R} \cup \text{outvars}_{R}) \& \text{wherep}) \\
&\quad [v/\text{outvars}_{R}] @ \text{pre} \\
&\quad \text{then} \\
&\quad \text{let } \text{outvars}_{R} \\
&\quad \text{be } \text{epreds}(\text{tdom}, \text{whenvars}_{R} \cup \\
&\quad \text{sourcevars}_{R} \cup \text{outvars}_{R}) \& \text{wherep} \\
&\quad \text{in } \text{CreateTrace}_{R} \\
&\quad \text{else} \\
&\quad \text{epreds}(\text{tdom}, \text{whenvars}_{R} \cup \\
&\quad \text{sourcevars}_{R}) \& \text{wherepx} \\
&\quad \text{endif}
\end{align*}
\]

where $Create\text{Trace}_{R}$ is $R\text{Strace}\rightarrow\text{exists}(\text{tr}|\&_{v \in \text{outvars}_{R}\text{tr}}. \\
sv = sv \& \&_{v \in \text{outvars}_{R}\text{tr}}tv = tv)$. 

To require that check-before-enforce semantics is used for a relation, a stereotype/annotation @checkBeforeEnforce could be written before the relation header. Alternatively, the stereotype could be written before a transformation header to express that it applies to all relations in the transformation. This definition of target element resolution affects the circumstances under which consistency conditions (b) and (c) are valid, as discussed in Sect. 6.

### 5.6.2 Least-change semantics

In some cases, it is useful to be able to specify that existing target elements should be reused and updated if they partially match the required conditions of a target pattern. The ‘check-before-enforce’ semantics only deals with situations where target elements completely match the required conditions, whilst the ‘always create new elements’ semantics ignores existing target elements. Key-based partial matching and update only operates for elements with keys.

We use the notation

\[ e <:: E \{ \text{pattern} \} \]

for target object templates where least-change partial matching is required. Such templates have the semantic interpretation $E \rightarrow \text{exists}LC(e|Pred)$ in the $\text{epreds}$ predicate, instead of $E \rightarrow \text{exists}(e|Pred)$, where $Pred$ semantically interprets $\text{pattern}$. Examples of the use of this mechanism are in the online dataset directories confluence and simplefamilies2persons of [27]. The application of the semantics to the families to persons case is explained in [26].

As for check-before-enforce, this definition of target element resolution alters the circumstances under which consistency conditions (b) and (c) are valid.

#### 5.6.3 Non-persistent traces

We have defined the semantics $\tau'$ of $\tau$ on the basis that trace instances of the $R\text{Strace}$ classes are persisted to support incremental execution. However, phases $Con_{\tau}(m)$ and $\text{Cleanup}_{\tau}(m)$ are also applicable in the case of non-persistent traces. The properties (1), (2), (3) remain valid as postcondition properties established at termination of $\tau'$, and the proof of Theorem 2 remains valid for batch-mode execution.

However, incremental execution mode is no longer supported. Execution of $\tau'$ on a non-empty target model may fail to propagate changes to the correct target objects, because the information of precise source–target correspondences in persistent traces used by $\text{Pres}_{\tau}(m)$ is not available. Each $Con_{\tau}(m)$ will be re-applied to every relevant source element, irrespective of previous mappings established for these elements via $R$.

#### 5.6.4 Propagation of element removal

The semantics we have presented does not necessarily propagate element removal changes. To address this, an explicit $\text{undo}_{R}(m)$ predicate can be derived, which is a conjunction of predicates $y.r \rightarrow \text{excludes}(x)$ for each predicate $y.r \rightarrow \text{includes}(x)$ or $x : y.r$ in $\text{epreds}(\text{tdom}, \text{outvars}_{R} \cup \text{sourcevars}_{R} \cup \text{whenvars}_{R}) \& \text{wherep}$, where $x$ and $y$ are in $\text{outvars}_{R}$. $\text{Pres}_{R}(m)$ is then modified to perform $\text{undo}_{R}(m)$ if the guard of $R$ is false for an existing trace:

\[
R\text{Strace}\@\text{pre} ::=
\begin{align*}
&\text{if } \text{guard}_{R}(m) \text{ then } \theta_{R}(m) \\
&\text{else } \text{undo}_{R}(m) \& \text{self} \rightarrow \text{isDeleted}() \\
&\text{endif}
\end{align*}
\]

Similarly for $\text{PresOp}_{R}$ in the case of non-top $R$.

#### 5.6.5 Internal composition of transformations

QVT-R transformations are typically written in a monolithic style, with all operations and rules contained in a single semantic unit. For large transformations (such as the RelToCore transformation, or the Ecore to SQL case of [36]), this can result in very large numbers of rules and operations (e.g. 51 operations, 12 rules and 118 calling dependencies in the first version of ecore2sql). To improve the organisation of large transformations, we suggest using subtransformations which are internally sequentially composed. The structure of such a transformation would be:
transformation t(pars)
{ common declarations of keys, shared operations

subtransformation t1(pars1)
{ relations for t1,
  operations used only in t1
}

...
implementing qvt-r via semantic interpretation in uml-rsd
• Entity splitting/merging: where the data of one class in $SL$ are distributed to data of two or more classes in $TL$, or vice versa. **Horizontal** merging/splitting is the situation where two or more exclusive classes are merged into/split out from one class. **Vertical** merging/splitting is the situation where two or more non-exclusive classes are merged into/split out from one class.

• Flattening/unflattening: various situations in which source model data structure corresponds to simplified or elaborated structure in the target model. For example, **introduce intermediate class** adds a new class $C$ in $TL$ between two classes derived from $SL$ linked by an association: $A \rightarrow_r B$ in $SL$ is elaborated to $A1 \rightarrow_{r1} C \rightarrow_{r2} B1$ in $TL$, so that $r$ is represented by the composition $r1r2$. The reverse of this transformation is a flattening which discards the intermediate elements. A **recursive *-accumulation** is a flattening where the closure of a source association corresponds to a target association.

Transformations involving flattening can be problematic for bidirectional execution because of the loss of information (e.g. the two cases where a bx is not definable in [36] both involve flattening).

• Auxiliary metamodel: Introduce additional model structure in order to support bidirectionalisation, e.g. to introduce flag attributes in target classes to record the specific source class from which a target instance is derived, in the case of horizontal Entity merging.

• Auxiliary models: in cases where there is substantial disparity between the structures of $SL$ and $TL$, introduce an intermediate model and split $\tau$ into a sequential composition of two transformations which use this model. Auxiliary models can also contain configuration information to restrict nondeterminism, as in the Families to Persons case [37].

• Object indexing: Introduce identity attributes/keys for elements in order to enforce reuse of target elements (instead of creation of new elements) and to enforce 1–1 or $n - 1$ relations between source and target elements of corresponding classes.

• Restrict input ranges: Use additional guards to prevent duplicated application of relations with source object
variables $s_1 : S, s_2 : S$—so that each pair $x_1, x_2$ of distinct $S$ elements is only bound in one way to the $s_i$,
similarly for other situations where repeated application
to the same input variables should be prevented. The additional guards should not involve relation tests.

In addition to design patterns, some specific idioms are useful:

- **Marker relation** [6]: simple non-top relations with domains $s : S(); t : T();$ with no explicit functionality, which are called from the *where* clause of more complex relations in order to store specific pairs or tuples of elements into a trace that can be queried in other relations. For example, the dag2ast/ast2dag and.ecore2sql transformation versions of [36] use this idiom.
- **Test and update bidirectional 1.* or 1-0..n associations at the 1 end:** as noted in Sect. 2, matching on 0..n multiplicity or * multiplicity association ends has semantic complications. Therefore, it is preferable to manipulate such associations via their opposite ends if these are 1-multiplicity.
- **Replace recursion by iteration:** recursive relations can be a cause of semantic problems and can be avoided in many
cases by using the Map Objects before Links pattern, by using the $\rightarrow$ closure($r$) operator on a self-association $r$ [28] or by using a $\forall$ quantifier (e.g. the bag migration case in Sect. 8, defined in “Appendix E”).

Deletion by selective copying is useful for update-in-place transformations (“Appendix A”).

8 Evaluation

In this section, we consider the cases of [36] and other bx examples, and use the above patterns and the implementation of QVT-R via UML-RSDS to provide systematic specifications of these. We consider both batch-mode and incremental-mode execution.

The code of all examples, together with their semantic interpretations in UML-RSDS and example execution scenarios, can be found at [27]. Due to space restrictions, we provide the detailed evaluation results in supplementary material (“Appendix E”). The Families to persons case is also presented as an example in [26]. We used Version 1.9 of the Agile UML tools for UML-RSDS, available at https://projects.eclipse.org/projects/modeling.agileuml. We used the current version of the Medini QVT tools [11]. All tests were carried out on a Windows 10 i5-6300 dual core laptop with 2.4GHz clock frequency, 8GB RAM and 3MB cache.

8.1 Comparison

The cases can be evaluated in terms of the quality metrics of [13,23] and in terms of the bx properties they support. Table 7 compares previous versions of the cases in QVT-R or ETL (for the tree to graph case) with the QVT-R and UML-RSDS versions defined in this paper, with regard to the number of quality flaws per LOC. Quality flaws are counts ERS and EHS of excessively large (over 50 LOC) rules/helpers, counts EFO and EPI of rules with excessive (over 5) fan-out and fan-in, counts EPL of excessive (over 10) numbers of variables, and counts CBR of excessive numbers of calling dependencies and DC of code clones. We also give the performance gain ratios of our version relative to the original version executed via Medini QVT (for the cases of [36]). In each case, our solutions have the same or fewer flaws than previous solutions and are more efficient for batch-mode execution. Table 8 summarises the bx properties of our solutions. Reduced flaws are due to our use of patterns, especially the Map objects before links pattern, which leads to a logical specification style with top relations dependent via when, rather than a functional style using non-top relations with where calls. Improved efficiency is due to optimisation in the UML-RSDS design synthesis process (e.g. reducing the range of element quantifications and searches where possible), and because code in 3GLs is produced, whereas Medini uses interpreted execution.

We have therefore improved on the properties of the solutions of [36] in two cases: the mapping of bags has been specified by a bidirectional transformation instead of by two separate forward and reverse transformations, and for trees and dags we specified forward and reverse transformations which are closely related and mutually inverse. We also provided incremental solutions for 4 of the 6 cases of [36] and provided a deterministic solution for the sets/ordered sets case. We improved the Hsm2nhsm transformation of [28] by eliminating the circular calling dependencies of the previous solution.

9 Related work

The closest related work to our approach is [2] and [8]. In [8] QVT-R is translated into coloured Petri nets (CPNs). This enables the use of CPN tools to simulate and analyse QVT-R specifications. The translation covers batch-mode execution but not incremental or in-place modes. They follow the semantic approach of RelToCore, using non-persistent trace. Separate CPN representations of source-to-target and target-to-source execution directions are generated. Both checkonly and enforce semantics are treated. Only check-before-enforce using key-based target resolution seems to be considered in [8]. As in our approach, read access to target models is considered an error in [8]. Criteria for termination are provided, as are consistency checks. However, general OCL expressions are not supported in specifications, and verification requires a developer to relate CPN-based analysis results to the original QVT-R text, which is not trivial, due to the complex and low-level encoding. Execution via CPN is possible; however, this is not efficient and is only useful for debugging/simulation. In [2], QVT-R is translated into a transition system formalism with model mu-calculus constraints expressing the semantics. Both batch and incremental mode are supported, but non-top relations are not permitted to create elements. The formal representation varies depending on the transformation direction. Key-based and check-before-enforce target resolution is adopted, without traces. As with [8], the formalism is quite distant from the UML/OCL basis of QVT-R, and it is non-trivial to relate results from the semantics back to the original specification. Execution and verification via model checking are supported in principle, but not implemented by [2]. Table 9 summarises the approaches of [2,8] and compares them with our approach. Our approach is the only one which formalises both least-change and in-place execution modes and provides 3GL code implementation. The conceptual distance of our formalisa-
Implementing QVT-R via semantic interpretation in UML-RSDS

Table 7: Quality flaw and performance measures for cases

| Case                  | Original version Flaws/LOC | Revised version Flaws/LOC | Performance gain |
|-----------------------|----------------------------|---------------------------|------------------|
| Tree to graph [14]    | 1/15                       | 0/17                      | –                |
| UML to Python         | –                          | 0/30                      | –                |
| Hsm2nhsm [28]         | 2/48                       | 1/70                      | –                |
| Person migration      | 0/63                       | 0/19                      | 893 (forward)    |
|                       |                            |                           | 575 (reverse)    |
| Weighted/unweighted   | 2/115                      | 1/60                      | 228 (forward)    |
| Petri nets            |                            |                           | 182 (reverse)    |
| Unordered/ordered sets| 1/121                      | 0/29                      | 563 (reverse)    |
| Migration of bags     | 1/157                      | 0/66                      | 45,404 (forward) |
| Gantt2CPM             | 10/378                     | 1/54                      | –                |
| Expression trees/dags | 8/439                      | 0/80                      | 8.1 (forward)    |
|                       |                            |                           | 34.5 (reverse)   |
| **Total flaws/LOC**   | **25/1336 (0.019)**        | **3/425 (0.007)**         |                  |

The problems with QVT-R semantics seem to go back to the original RFP for the language, which emphasised technology alignment instead of language semantics [15]. Specific problems in QVT-R semantics were documented by Stevens in [34]. The limitations of a purely declarative interpretation of QVT-R were identified. This work led to the semantics of QVT-R defined using mu-calculus [2]. An alternative QVT-R semantics is defined in [28], using constraint solving in alloy to implement a least-change interpretation of relations in enforce mode. We have instead formulated a least-change semantics within standard UML/OCL.

Other approaches concern implementation of QVT-R by translation to another language or formalism, but do not provide semantic analysis. For example, [32] translates QVT-R to QVT-O, and [7] translates QVT-R to TGG. The translation to TGG, however, helps to expose semantic ambiguities in the QVT-R semantics. In [39], a fine-grain model of QVT-R computations is defined and used to guide automated execution optimisation. Here we have described how QVT-R specifications can be restricted and organised to avoid self-dependencies of relations and mutual dependencies between relations, and other forms of circular data-dependence. This enables a simpler implementation approach to be adopted whilst retaining high efficiency.

In [36], the problems in defining bx in QVT-R are investigated via seven case studies. Specific limitations of QVT-R are identified: the lack of definite ordering in where clause/target domain execution, which complicates the definition of bx execution strategies, and the lack of facilities for defining variability options, such as rules specific to one execution direction. We have addressed the first problem in our semantics. The second would be a useful facility, especially in cases where most relations can operate in both forward and reverse directions, but some are only valid in one direction. The transformation could be divided into three subtransformations: one ($\tau$) for the fully bx relations, one ($\sigma$) for the specific forward relations and another ($\rho$) for the specific reverse relations. Transformation extension could be used to combine $\tau$ with $\sigma$ and $\tau$ with $\rho$.

Additionally, [36] identifies tooling problems for QVT-R, in that the only practical QVT-R tool available, Medini QVT [11], does not support the full language. We have defined tooling support which overcomes this problem via translation to UML-RSDS (effectively a subset of UML) and its tools. This supports domain conditions, relation overriding and transformation extension (unlike Medini) and provides an option to use least-change semantics.

Similar work has been carried out for other MT languages. For example, [4] translate ATL into an intermediate representation with a formal semantics to support verification. Similarly, [35] translate ATL to the Maude formalism. Operationalisation approaches for TGG [5,9] have some similarities to our approach, with the correspondence models in TGG being used in a similar way to QVT-R traces. However, TGG is a simpler language than QVT-R because TGG rules cannot explicitly refer to other rules, and the expression language used for TGG is simpler than OCL. As in our approach, [9] also uses multiple phases to define an incremental operational form of a TGG specification. However, this form is a graph transformation, not a logical representation, and requires the introduction of additional flag attributes. In [25], we define an approach for bidirectionalising UML-RSDS using syntactic inversion of predicates. Although this produces efficient transformations, it requires the specifier to ensure that they remain within the subset of the language which is syntactically invertible. Using QVT-R enables bx to
Table 8  Bx properties for cases

| Case                | Bidirectionality | Batch/incremental | Deterministic | Patterns used                                                                 |
|---------------------|------------------|-------------------|---------------|-------------------------------------------------------------------------------|
| Tree to graph       | Bidirectional    | Incremental       | Yes           | Introduce/remove intermediate class                                            |
|                     |                  |                   |               | Map objects before links                                                       |
|                     |                  |                   |               | Recursive *-accumulation                                                       |
|                     |                  |                   |               | Auxiliary metamodel                                                           |
|                     |                  |                   |               | Map objects before links                                                       |
| UML to Python       | Bidirectional    | Incremental       | Yes           | Recursive *-accumulation                                                       |
|                     |                  |                   |               | Entity merging/splitting (horizontal)                                           |
|                     |                  |                   |               | Map objects before links                                                       |
|                     | Partly separate  | Incremental       | Yes           | Recursive *-accumulation                                                       |
|                     | forward/reverse  |                   |               | Entity merging/splitting (horizontal)                                           |
|                     |                  |                   |               | Map objects before links                                                       |
| Hsm2nhsm            |                  | Incremental       | Yes           | Recursive *-accumulation                                                       |
| Person migration    | Bidirectional    | Incremental       | Yes           | Lens                                                                          |
| Weighted/unweighted | Bidirectional    | Incremental       | Yes           | Introduce/remove intermediate classes                                          |
| Petri nets          |                  |                   |               | Map objects before links                                                       |
| Unordered/ordered   | Bidirectional    | Incremental       | Yes           | Map objects before links                                                       |
| sets                |                  |                   |               | Object indexing                                                               |
| Migration of bags   | Bidirectional    | Batch             | Yes           | Auxiliary models                                                              |
|                     |                  |                   |               | Object indexing                                                               |
|                     |                  |                   |               | Lens; Introduce/remove intermediate class; Map objects before links            |
|                     |                  |                   |               | Deletion by selective copy                                                    |
|                     | Mutually inverse | Batch             | No            | Entity merging/splitting (horizontal)                                           |
|                     | forward/reverse  |                   |               | Introduce/remove intermediate class                                            |
|                     | Bidirectional    | Incremental       | Yes           | Map objects before links                                                       |

be defined in many cases using a single specification for forward and reverse directions, avoiding the need for syntactic bidirectionalisation. We have extended the QVT-R semantics of [25] to provide variations in target element resolution semantics, including the Medini QVT semantics and update-in-place semantics.

10 Conclusions

In this paper, we have shown that a coherent mathematical semantics can be given to QVT-R, based on a translation to UML-RSDS. This semantics is compatible with the de facto QVT-R semantics given by the Medini QVT tool, and it is consistent with the (partial) QVT-R to QVT-C translation of [30].
The semantics provides a basis for the static analysis of QVT-R specifications and for the identification of semantic problems in them. We provide variation points to permit alternative target resolution approaches and trace models. In addition, by translating to UML-RSDS, specifiers gain the ability to perform other semantic checks, such as confluence analysis, and to generate efficient implementations of QVT-R specifications in multiple programming languages. By adopting MT design patterns from UML-RSDS, it also becomes possible to systematically construct QVT-R bx of particular kinds. We consider that overall our approach can contribute to increasing the precision of QVT-R semantics and enhancing the usability of the language for practitioners.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix A: Semantic translation for update-in-place transformations

The definition of the semantics of update-in-place transformations in [30] is very brief and lacks detail (Section 7.7 of [30]). We can adapt the separate-models mode semantics of Sect. 5.2 to provide a translation for in-place QVT-R transformations.

Update-in-place transformations τ operate on a single model m : L, which is both the source and target of τ. All domains of relations R have model m, and two domains may have root variables e : E with the same name. An element e bound to these variables is therefore both read and written by R. In this case, the first of these domains can be designated as a source and the second as a target:

```
top relation R
{ checkonly domain m e : E { ...
  source domain ... };}
enforce domain m e : E { ...
  target domain defining update of e ... };
when { ... }
where { ... }
}
```

$sdom$ is thus the set of checkonly domains, and $tdom$ the set of enforce domains. For update-in-place $\tau$, we only consider unidirectional top relations. In the case of multiple domain root variables $e : E$ with the same names and types, traces for $R$ record only a single $e : E$ property.
\( \theta_R(m, \text{vars}) \) is defined as for the separate-models case, but \( Pres_R(m) \) is redefined as:

\[
R$\text{trace}@pre ::
\begin{align*}
  & \text{if guard}_R(m)@pre \text{ then } \theta_R(m)' \\
  & \text{else } \text{self} \rightarrow \text{isDeleted}()
\end{align*}
\]

where \( \theta_R(m)' \) has antecedent \((\text{cpreds}(\text{sdom}, \text{ovars}_R) \& \text{whenp}@pre)@pre\). We use this prestate version of the antecedent of \( \theta_R \) for the update-in-place semantics because the same data (features of \( E \)) may be read and written by \( R \). The use of the prestate expression enforces a bounded-loop implementation, which does not necessarily ensure that the relation is established for all applicable instances of \( E \). Instead, the entire transformation will be iterated until all top relations are simultaneously established (as in Section 7.7 of [30]).

\( Con_R(m) \) is modified in a similar way to refer to prestate versions of data on the LHS:

\[
\quad::
\begin{align*}
  & (\text{cpreds}(\text{sdom}, \{\}) \& \text{whenp}@pre \& \\
  & \text{not}(R$\text{trace}@pre \rightarrow \text{exists}(tr | tr.e = e)) \Rightarrow
\end{align*}
\]

\[
\text{epreds}(\text{tdom}, \text{whenvars}_R \cup \text{sourcevars}_R) \& \text{wherepx}
\]

The \( \text{Cleanup}_E \) constraints are defined as in the separate-models case, except that the \( Ri \) range over all relations with some \( e : E \) variable in their \( \text{tvars}_R \) set (instead of \( \text{outvars}_R \)). The reason for this change is that target elements can exist because they were copied from the source model, and do not need to be explicitly created. Unlike the separate-models case, the \( \text{Cleanup}_E \) constraints can delete source elements (since the source and target models are the same) and delete traces.

The phases \( \text{Pres}_r(m), \text{Con}_r(m), \text{Cleanup}_r(m) \) are then defined from the \( \text{Pres}, \text{Con} \) and \( \text{Cleanup} \) constraints as in the separate-models case. Correctness conditions (b) to (e) are also applicable to update-in-place transformations, but condition (a) is weakened to:

- (a’) Relations can only refer to target features/class names via explicitly declared object variables \( t : T \) and their features.

Two cases of update-in-place transformation can be distinguished: (i) where a single execution is sufficient to establish the required target model; (ii) where repeated iteration is necessary. We suggest using transformation stereotypes/annotations to distinguish these cases. For iterated transformations, an annotation @iterated could be written before the transformation header to indicate that the transformation should have this semantics.

A simple example of an update-in-place rule for the UML metamodel of Fig. 3 is:

\[
\text{top relation } R
\]

\[
\begin{align*}
  & \{ \text{checkonly domain design } e : \text{Entity} \}; \\
  & \text{enforce domain design } e : \text{Entity} \{
  & \{ e.\text{stereotypes} \rightarrow \text{includes}("\text{COM+}") \};
  \\
  & \}
\end{align*}
\]

For this rule, \( Pres_R(\text{design}) \) and \( Con_R(\text{design}) \) are:

\[
R$\text{trace}@pre::
\begin{align*}
  & \text{if } e : \text{Entity}@pre \text{ then }
  \end{align*}
\]

\[
\begin{align*}
  & e.\text{stereotypes} \rightarrow \text{includes}("\text{COM+}")
  \text{else}
  & \text{self} \rightarrow \text{isDeleted}()
  \text{endif}
\end{align*}
\]

\[
::
\begin{align*}
  & e : \text{Entity}@pre \& \text{not}(R$\text{trace}@pre \rightarrow \text{exists}(tr | tr.e = e)) \Rightarrow
  \\
  & e.\text{stereotypes} \rightarrow \text{includes}("\text{COM+}") \&
  \\
  & R$\text{trace}@exists
  \begin{align*}
  & \{ tr | tr.e = e \}
  \end{align*}
\]

This transformation does not need to be iterated, because after one iteration it ensures that \( \theta_R(\text{design}) \) is true for all model instances. Notice that attributes such as \text{name} and \text{isAbstract} do not need to be explicitly copied. More complex update-in-place examples are transformations such as class diagram refactoring [12] or graph rewriting with implicit element deletion:

\[
\text{top relation } R
\]

\[
\begin{align*}
  & \{ \text{checkonly domain graph } n : \text{Node} \}; \\
  & \text{enforce domain graph } n : \text{Node} \{
  & \{ n.\text{neighbours} \rightarrow \text{size}() = 1 \}
  \\
  & \}
\end{align*}
\]

where graph nodes have a set \text{neighbours} : \text{Set(Node)} of neighbours, a \text{name} : \text{String} and a unique \text{id} : \text{String}. Such transformations may require repeated execution. In this example, \( Con_R(\text{graph}) \) will not be enabled on \( n \) which have \( n.\text{neighbours}@pre \rightarrow \text{size}() \neq 1 \); hence, such nodes will not be recorded in \( R$\text{trace} \) and will be deleted by \text{CleanupNode}(\text{graph}) \). This implicit deletion may then reduce the set of neighbours of other nodes and result in \( R \) being enabled on these in the next iteration.

\[
\text{Con}_R(\text{graph}) \text{ is:}
\]

\[
::
\begin{align*}
  & n : \text{Node}@pre \& n.\text{neighbours}@pre->
\end{align*}
\]
Appendix B: Definition of cpreds and epreds

cpreds and epreds convert the sequences of source and target relation domains of a QVT-R relation into UML-RSDS OCL predicates. These use auxiliary functions cpredlepred on individual relation domains.

cpreds([], bounds) is true, whilst cpred([d] ^ doms, bounds) is \( p \land \text{cpred}(d.\text{rootVariable}, d.\text{pattern.templateExpression}, \text{bounds} \cup \{d.\text{rootVariable}\}) \) and \( \text{cpred}(e, \text{expr}, \text{bound}) \) interprets a template or OCL expression \( \text{expr} \) as a UML-RSDS OCL expression as follows, where \( \text{bound} \) is the set of variables which are bound at the current text point.

In the case of a primitive domain, \( e : T, \text{expr is true} \) and \( \text{cpred}(e, \text{true}, \text{bound}) \) is simply \( (e : T, \text{bound} \cup \{e\}) \).

If \( \text{expr} \) is an \( \text{ObjectTemplateExp} \) of the form

\[
e : E \{ f_1 = v_{a_1}, \ldots, f_n = v_{a_n} \} \{ P \}
\]

then the first result \( p \) of \( \text{cpred}(e, \text{ote}, \text{bound}) \) is formed as a conjunction of the predicates of \( \text{cpred}(e, f_i = v_{a_i}, \text{bound}) \) for each \( \text{PropertyTemplateItem} f_i = v_{a_i}, \text{where} \) \( \text{bound}_1 = \text{bound} \cup \{e\} \):

- If \( \text{vali} \) is itself an \( \text{ObjectTemplateExp} \), such as \( x : F \{ g_1 = w_{1,1}, \ldots, g_m = w_{1,m} \} \), then the predicate of \( \text{cpred}(e, f_i = v_{a_i}, \text{bound}) \) is \( x = e.f_i \land x : F \) conjoined with the predicate result \( q \) of \( \text{cpred}(x, v_{a_i}, \text{bound} \cup \{x\}) \) if \( f_i \) is single-valued (its multiplicity is 1), or \( x : e.f_i \land x : F \land q \) if \( f_i \) is collection-valued (multiplicity \( \neq 1 \)). The type declaration \( x : F \) is omitted if \( x \in \text{bound}_1 \), since it will already have been previously asserted.

\( x \) is considered to be bound by the first conjunct in both cases if it is not already bound. That is, the bound set result \( \text{bound}_{i+1} \) of \( \text{cpred}(e, f_i = v_{a_i}, \text{bound}) \) includes \( x \) and \( \text{bound}_1 \).

- Otherwise, if \( \text{vali} \) is an unbound variable, \( \text{vali} \notin \text{bound}_1 \), and \( f_i \) is single-valued, \( \text{cpred}(e, f_i = v_{a_i}, \text{bound}) \) is \( (\text{vali} = e.f_i, \text{bound} \cup \{\text{vali}\}) \). If \( f_i \) is collection-valued, the predicate is \( \text{vali} = e.f_i, \text{bound} \cup \{\text{vali}\} \), \( \text{bound}_{i+1} = \text{bound}_1 \cup \{\text{vali}\} \).

- If \( \text{vali} \) is a bound variable or other expression, the result is \( (e.f_i = \text{vali}, \text{bound}) \) for single-valued \( f_i \) or \( (\text{vali} : e.f_i, \text{bound}) \) for collection-valued \( f_i \). That is, \( \text{bound}_{i+1} = \text{bound}_1 \).

In the case of a set-typed collection template expression \( \text{cte} \):

\[
s : \text{Set}(E) \{ x \land \text{rest} \}(P)
\]
The root variable $e$ of the target domain is $\text{exists}$-quantified or $\text{existsLC}$-quantified over its type, if it is not already bound. Any additional predicate $P$ of the domain is added as the final conjunct of the predicate of $\text{epred}(e : E[], P, \text{bound})$. The $\text{exists}$ or $\text{existsLC}$ quantifiers introduced in $\text{epred}$ have their scope extended over the remainder of the logical interpretation of $\text{Pres}_R$, including the $\text{where}$ predicate interpretation\(^4\).

**Appendix C: Definitions of $\text{rd}$, $\text{wr}$, $\text{stat}$, $\text{stat}_{\text{LC}}$**

### C.1 Read and write frames

Table 10 (an updated version of the corresponding table of [24]) gives some cases of the definitions of read and write frames of OCL constraints. $\text{var}(P)$ is the set of all features and entity type names used in $P$, likewise for $\text{var}^*(P)$. $V \downarrow v$ is $V$ with pairs $(x, f)$ removed, where $x \in v$.

In computing $\text{wr}(P)$, we also take account of the features and entity types which implicitly depend upon the explicitly updated features and entity types of $P$, such as inverse association ends. In particular, if an association end role2 has a named opposite end role1, then role1 depends on role2 and vice versa. Creating an instance $x$ of a concrete entity type $E$ also adds $x$ to each supertype extent $F$ of $E$, and so the extents of these supertypes are also included in the write frames of $E \rightarrow \text{exists}(x|Q)$ and $E \rightarrow \text{existsLC}(x|Q)$ in Table 10.

Deleting an instance $x$ of entity type $E$ by $x \rightarrow \text{isDeleted}()$ may affect any supertype of $E$ and any association end owned by $E$ or its supertypes, and any association end incident with $E$ or with any supertype of $E$. Additionally, if entity types $E$ and $F$ are related by an association which is a composition at the $E$ end, or by an association with a mandatory multiplicity at the $E$ end, i.e. a multiplicity with lower bound 1 or more, then deletion of $E$ instances will affect $F$ and its features and supertypes and incident associations, recursively. $\text{del}(x)$ and $\text{del}^*(x)$ are the corresponding sets of features and class names potentially written by deletion propagation from $x$.

$\text{wr}(G)$ of a set $G$ of constraints is the union of the constraint write frames, likewise for $\text{rd}(G)$, $\text{wr}^*(G)$, $\text{rd}^*(G)$.

### C.2 Design synthesis

The design-level activity $\text{stat}(P)$ synthesised from an effective-for-update OCL predicate $P$ is defined systematically based on the structure of $P$. $\text{stat}(P)$ can be read as ‘Make $P$ true’. $P$ involving negations $\text{not}(Q)$ are normalised where possible so that $\text{not}$ is removed, e.g. $\text{not}(s \rightarrow \text{includes}(x))$ is rewritten to $s \rightarrow \text{excludes}(x)$.

Table 11 shows the main cases of the $\text{stat}$ definition (extended and refined from [24]). In the context of QVT-R semantics, an expression is assignable if it is a local domain variable, or is of the form $v.f$ for a feature $f$ of a target

---

\(^4\) Using the combination rule of $E \rightarrow \text{exists}(x|P)$ and $F \rightarrow \text{exists}(y|Q)$ to $E \rightarrow \text{exists}(x|P \& F \rightarrow \text{exists}(y|Q))$ recursively, and similarly for $\rightarrow \text{existsLC}$. 

---

\(\ominus\) Springer
Table 10 Definition of read and write frames

| P                                                                 | \( rd(P) \)          | \( wr(P) \) | \( rd^*(P) \)        | \( wr^*(P) \)        |
|------------------------------------------------------------------|----------------------|-------------|----------------------|----------------------|
| Basic expression \( e \) without quantifiers, logical operators or \( ==, \neq, \_\_\_, \leq, E[] \) | Set of features and class names used \( in\ P: var(P) \) | \( [\_\_\_] \) | Set of pairs \( (obj, f) \) of objects and features, \( obj.f \), in \( P, plus \) class names in \( P \): \( var^*(P) \) | \( [\_\_\_] \) |
| \( e1 : e2.r \)                                                 | \( var(e1) \cup var(e2) \) | \( [r] \)    | \( var^*(e1) \cup var^*(e2) \) | \( [(e_2, r)] \) |
| \( e2.r \rightarrow \text{includes}(e1) \)\( e1, e2 \) single-valued \( r \) | \( var(e1) \cup var(e2) \) | \( [r] \)    | \( var^*(e1) \cup var^*(e2) \) | \( [(e_2, r)] \) |
| \( e1 / : e2.r \) \( e1, e2 \) single-valued \( r \)           | \( var(e1) \cup var(e2) \) | \( [f] \)    | \( var^*(e1) \cup var^*(e2) \) | \( [(e_1, f)] \) |
| \( e2.r \rightarrow \text{excludes}(e1) \) \( e1 \) \( e2 \) single-valued \( r \) | \( var(e1) \cup var(e2) \) | \( [r] \)    | \( var^*(e1) \cup var^*(e2) \) | \( [(e_2, r)] \) |
| \( E[1] \) \( E \rightarrow \text{exists}(x|Q) \) \( E \) single-valued \( x \) \( x \) single-valued, of class type \( E \) | \( var(e1) \cup \{E\} \) | \( [\_] \) | \( var^*(e1) \cup \{E\} \) | \( [\_] \) |
| \( E \rightarrow \text{exists}(x|Q) \) \( E \) \( E \rightarrow \text{existsLC}(x|Q) \) \( E \) \( E \rightarrow \text{forall}(x|Q) \) \( E \) \( E \rightarrow \text{isDeleted}() \) | \( rd(Q) \) | \( wr(Q) \cup \{E\} \cup \text{all superclasses of} E \) | \( rd^*(Q) \) | \( wr^*(Q) \cup \{E\} \cup \text{all superclasses of} E \) |
| \( x \rightarrow \text{isDeleted}() \) \( x \) single-valued, of class type \( E \) | \( rd(Q) \cup \{E\} \) | \( wr(Q) \) | \( rd^*(Q) \cup \{E\} \) | \( wr^*(Q) \) |
| \( Q \rightarrow R \) \( Q \) \( Q \) \( Q \times R \) \( Q \) \( if \ E \) \( E \) \( else \ E2 \) | \( var(C) \cup rd(Q) \) | \( wr(Q) \) | \( var^*(C) \cup rd^*(Q) \) | \( wr^*(Q) \) |
| \( Q \times R \) \( Q \) \( if \ E \) \( E \) \( else \ E2 \) | \( rd(Q) \cup rd(R) \) | \( wr(Q) \cup wr(R) \) | \( rd^*(Q) \cup rd^*(R) \) | \( wr^*(Q) \cup wr^*(R) \) |
| \( Q \) \( Q \rightarrow R \) \( Q \) \( Q \) \( Q \times R \) \( Q \) \( if \ E \) \( E \) \( else \ E2 \) | \( rd(Q) \cup rd(R) \) | \( wr(Q) \cup wr(R) \) | \( rd^*(Q) \cup rd^*(R) \) | \( wr^*(Q) \cup wr^*(R) \) |
| \( Q \) \( Q \rightarrow R \) \( Q \) \( Q \) \( Q \) \( Q \) \( if \ E \) \( E \) \( else \ E2 \) | \( var(E) \cup wr(E1) \cup wr(E2) \) | \( var^*(E) \cup wr^*(E1) \cup wr^*(E2) \) | \( rd^*(E1) \cup rd^*(E2) \) | \( wr^*(E2) \) |
| \( let \ v \in P \) \( P \) \( let \ v \in P \) \( P \) \( let \ v \in P \) \( P \) \( let \ v \in P \) \( P \) \( let \ v \in P \) \( P \) | \( \text{if} \ E \) \( E \) \( else \ E2 \) \( E2 \) | \( rd(Q) \cup \{v\} \cup \text{if} \ E \) \( E \) \( else \ E2 \) \( E2 \) | \( rd^*(Q) \cup \{v\} \cup \text{if} \ E \) \( E \) \( else \ E2 \) \( E2 \) | \( rd^*(Q) \cup \{v\} \cup \text{if} \ E \) \( E \) \( else \ E2 \) \( E2 \) |
sequences it is createBy P K E

\[ P \text{ holds.} \]

\[ \text{if } C \text{ then statLC}(P) \text{ else statLC}(Q) \text{ endif} \]

\( \exists \text{ deleted} \)

\( \text{createE()} \)

\( \text{isDeleted} \)

\( \text{existsLC} \)

\( \text{isEmpty} \)

\( \text{select} \)

\( \text{any} \)

\( \text{allInstances}() \)

\( \text{= value} \)

\( \text{id} \)

\( \text{pk} \)

\( \text{value} \)

\( \text{pk} = \text{value} \)

\( \text{createByPKE}() \)

\( \text{createE()} \)

\( \text{forall} \)

\( \text{existsLC} \)

\( \text{forall} \)
Table 11 Definition of stat(P)

| P                     | stat(P)                        | Condition                                      |
|-----------------------|--------------------------------|------------------------------------------------|
| true                  | skip                           | x is assignable,                               |
| x = e                 | x:=e                           | x * is assignble,                              |
| e : x                 | x:=x including(e)              | x is assignable,                               |
| x->includes(e)        |                                | collection-valued, x * is assignble,           |
| e / : x               | x:=x excluding(e)              | x is assignable,                               |
| x->excludes(e)        |                                | collection-valued, x * is assignble,           |
| e <<: x               | x:=x union(e)                  | x is assignable,                               |
| x->includesAll(e)     |                                | collection-valued, x * is assignble,           |
| e / <<: x             | x:=x - e                       | x is assignble,                                |
| x->excludesAll(e)     |                                | collection-valued, x * is assignble,           |
| x->isDeleted()        | -composition of                | Each class E                                   |
| (single object x)     | E:=E excluding(x)              | containing x                                   |
|                       | and y.r:=y.r excluding(x)      | each association end                           |
|                       | obj.op(e)                      | Single object obj                              |
|                       | obj expiry(e)                  | Collection obj                                 |
|                       | for x : objs do x.op(e)        |                                                |
| P1 & P2               | stat(P1); stat(P2)             | wr(P2) intersect wr(P1) = {}                   |
|                       |                                 | wr(P2) intersect rd(P1) = {}                   |
| P1 xor P2             | stat(P1); if not(P1) then stat(P2) else skip | not(P2) can be normalised                      |
|                       | if P1 & P2 then stat(not(P2)) else if not(P1) & not(P2) then stat(P1) else skip |                                                |
| E->exists(x | x.id = v & P1) | if E.id->includes(v) then stat(P1) else (var x : E := E[v]; stat(P1)) | E is a concrete class with E->isUnique(id)     |
|                       |                                |                                                |
| E->exists(x | P1) | (var x : E := E[v]; stat(P1)) | E is a concrete class, P1 not of form x.id = v & P2 for unique id attribute of E |
|                       |                                |                                                |
| e->exists(x | x.id = v & P1) | if e->includes(E[v]) then (var x : E := E[v]; stat(P1)) else skip | Non-writable expression e with element type class E, E->isUnique(id) |
|                       |                                |                                                |
| e->exists(x | P1) | if e->notEmpty() then (var x : E := e->any(true); stat(P1)) else skip | Non-writable expression e, P1 not of above form |
|                       |                                |                                                |
| E->forAll(x | P1) | for x : E do stat(P1) | wr(P1) intersect rd(P1) = {}                   |
| P1 => P2             | if P1 then stat(P2) else skip | P1 side-effect free                            |
|                       |                                 | wr(P2) intersect var(P1) = {}                  |
| if E then P1         | if E then stat(P1)             | E side-effect free                              |
| else P2 endif        | else stat(P2)                  |                                                |
| let v be P in Q      | any v where P then stat(Q)     | (wr(P1) union wr(P2)) intersect var(E) = {}     |
|                       |                                 | wr(Q) intersect var(P) = {}                    |
if eset->isEmpty()
  then
      statLC(Pk & ... & Pn)
  else
      ... case for k+1 ...

For n, if \( P_k \) is an assignment result = e, then the code of the case is simply \( e := \text{eset} \rightarrow \text{any} \{ \text{true} \}; \text{result} := e \). Otherwise it is

\[
e := \text{eset} \rightarrow \text{any} \{ \text{true} \};
\]

\[
\text{eset} := \text{eset} \rightarrow \text{select}(Pn);
\]

\[
e := \text{eset} \rightarrow \text{any} \{ \text{true} \}
\]

By reusing \( e : E \) instances where possible, the redundant creation of instances is avoided; however, this also introduces the possibility of conflicts where one target instance is required to have conflicting attribute values to satisfy a constraint wrt two source instances.

### Appendix D: Relation overriding and transformation extension

If relation \( R \) is declared as overriding relation \( S \), then both must either be top level, or both non-top level. \( R \) should have corresponding relation and primitive domains \( d_R \) for each relation and primitive domain \( d_S \) of \( S \), with the same name, model and modality:

\[
d_S \in S.\text{domain} \implies \exists d_R \in R.\text{domain} : \begin{align*}
  d_R.\text{rootVariable}.\text{name} &= d_S.\text{rootVariable}.\text{name} \\
  d_R.\text{isCheckable} &= d_S.\text{isCheckable} \\
  d_R.\text{isEnforceable} &= d_S.\text{isEnforceable} \\
  d_R.\text{isPrimitive} &= d_S.\text{isPrimitive} \\
  d_R.\text{typedModel} &= d_S.\text{typedModel}
\end{align*}
\]

The type \( d_R.\text{rootVariable}.\text{type} \) of \( d_R \) should be the same as that of \( d_S \) or a subclass/descendant of the \( d_S \) type. \( R \) may also have additional domains to those of \( S \). The common-named domains of \( S \) and \( R \) should occur in the same order in both relations. For update-in-place \( R \), \( S \), domains of \( S \) with the same root variable name are overridden by the corresponding domains of \( R \) on the basis of their modality.

The combination of \( S \) and \( R \) is expressed as a composed relation \( P = S \oplus R \). This has domains \( d_S \oplus d_R \) for common-named domains \( d \) of \( S \) and \( R \), together with any additional domains of \( R \). The \( \text{when} \) clause of \( P \) is the conjunction \( S.\text{when} \) and \( R.\text{when} \), i.e. the pattern formed by concatenating \( S.\text{when}.\text{predicate} \) and \( R.\text{when}.\text{predicate} \) and removing duplicated conjuncts. The same applies for the \( \text{where} \) clauses. \( P \) is abstract if \( R \) is abstract, and concrete if \( R \) is concrete.

\[
d_S \oplus d_R \text{ is defined by recursion on the structure of the domain templates. For object template expressions } t_S = d_S.\text{pattern}.\text{templateExpression}, t_R = d_R.\text{pattern}.\text{templateExpression}, \text{we can consider } t_S.\text{part} \text{ and } t_R.\text{part} \text{ to be ordered so that the common-named variables of the parts are listed together in the same order at the start of each } \text{part} \text{ sequence. For two parts on a common property of non-object type, } \oplus \text{ is defined on } \text{PropertyTemplateItem} \text{ to discard the first part:}
\]

\[
v = \text{val1} \oplus p = p
\]

where \( \text{p.preferredProperty.name} = v \).

Otherwise, if both parts are object definitions of same-named properties, the definitions are merged:

\[
f = v : E[p1] \oplus f = w : F[p2] = f = u : G[p3]
\]

where

\[
v : E[p1] \oplus w : F[p2] = u : G[p3]
\]

Object templates can be combined in this manner if \( w.\text{name} = v.\text{name} \) and either \( F = E \) or \( F \) is a subclass/descendant of \( E \). The result has \( u = w \) and \( G = F \).

Parts that belong to either \( t_S \) or \( t_R \) and have no corresponding part (with the same property name) in the other template are retained in \( t_S \oplus t_R \). The \( \text{where} \) conditions of the two templates are conjoined to form the \( \text{where} \) condition of the result.

The combination of two set-typed collection template expressions \( cte_S \):

\[
s : \text{Set}(E)\{x ++ \text{rest}\}\{P\}
\]

and \( cte_R \):

\[
s : \text{Set}(F)\{x ++ \text{rest}\}\{Q\}
\]

is defined as \( cte_S \oplus cte_R \):

\[
s : \text{Set}(F)\{x ++ \text{rest}\}\{P \text{ and } Q\}
\]

where \( F \) is \( E \) or a subclass/descendant of \( E \), likewise for sequence-typed collection templates.

Errors may arise if \( R \) and \( S \) contain same-named domains or same-named variables with conflicting types. For example, object variables \( x : E, x : F \) where \( E \) and \( F \) are...
not related by inheritance. Error messages are produced in such cases. If $S$ is called in a when or where clause, then the domain.rootVariable.name sequences of the two relations should be the same.

Transformation extension is syntactically represented as $\tau$ extends $\sigma$ in [30], but no semantics is provided. We can infer that $\tau$ and $\sigma$ should have the same set of typed models:

$$\tau.modelParameter = \sigma.modelParameter$$

The helpers of the combination $\rho = \mathit{\tau extends \sigma}$ of the transformations are the union of the helper sets of each transformation. Name clashes are not permitted:

$$\rho.helpers = \tau.helpers \cup (\sigma.helpers)$$

The relations of $\rho$ are the union of those of $\tau$ and $\sigma$, but with common-named rules of the two transformations combined using $\oplus$.

$$\rho.rule = \sigma.rule \cup \mathit{\tau.rule name \rightarrow includes (r.name)} \cup \mathit{\tau.rule name \rightarrow includes (r.name)} \cup \mathit{\sigma.rule \rightarrow select (r) \rightarrow \tau.rule name \rightarrow includes (r.name)} \cup \mathit{\tau.rule \rightarrow any (name = r.name))}$$

References

1. Anjorin, A., et al. A.: Benchmarx reloaded: a practical benchmark framework for bidirectional transformations, BX (2017)
2. Bradfield, J., Stevens, P.: Enforcing QVT-R with mu-calculus and games. FASE’13, LNCS (2013)
3. bx-community.wikitod.com/examples.uml2c
4. Chen, Z., Monahan, R., Power, J.: A sound execution semantics for ATL via translation validation, ICMT (2015)
5. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. SoSyM 8, 21–43 (2009)
6. Goldschmidt, T., Wachsmuth, G.: Refinement transformation support for QVT relational transformations, ENCS (2011)
7. Greener, J., Kindler, E.: Comparing relational model transformation technologies: implementing QVT with triple graph grammars. SoSyM 9(1), 21–46 (2010)
8. Guerra, E., de Lara, J.: Colouring: execution, debug and analysis of QVT-R transformations through coloured Petri nets. SoSyM 13(4), 1447–1472 (2017)
9. Herman, F., et al.: Model synchronisation based on triple graph grammars. SoSyM 14(1), 1–29 (2015)
10. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems, MODELS (2006)
11. IKV technologies, Medini QVT, projects.ikv.de/qvt/downloads. Accessed Dec. 2019
12. Kolahdouz-Rahimi, S., Lano, K., Pillay, S., Troya, J., Van Gorp, P.: Evaluation of MT approaches for model refactoring. Sci. Comput. Prog. 85(Part A), 5–40 (2014)
13. Kolahdouz-Rahimi, S., Lano, K., et al.: A comparison of quality flaws and technical debt in model transformation specifications. JSS (2020). https://doi.org/10.1016/j.jss.2020.110684
14. Kolovos, D., Paige, R., Polack, F.: The Epsilon Transformation Language, ICMT (2008)
15. Kurtev, I.: State of the art of QVT: a model transformation language standard, AGTIVE 2007. LNCS 5088, 377–393 (2008)
16. Kusel, A., Schwinger, W., Wimmer, M., Retschitzegger, W.: Common pitfalls of using QVT-Relations. ICECCS (2009)
17. Lano, K.: The B Language and Method. Springer, Berlin (1996)
18. Lano, K., Kolahdouz-Rahimi, S.: Constraint-based specification of model transformations. J. Syst. Softw. 86, 412–436 (2013)
19. Lano, K.: Agile Model-Based Development Using UML-RSDS. CRC Press, Boca Raton (2016)
20. Lano, K., Kolahdouz-Rahimi, S.: Model-transformation design patterns. IEEE Trans. Softw. Eng. 40(12), 1224–1259 (2014)
21. Lano, K., et al.: A framework for MT verification. FACS (2014)
22. Lano, K., Yassipour-Tehrani, S., Alfraihi, H., Kolahdouz-Rahimi, S.: Translating from UML-RSDS OCL to ANSI C, OCL (2017)
23. Lano, K., Kolahdouz-Rahimi, S., Sharbaf, M., Alfraihi, H.: Technical debt in Model Transformation specifications. ICMT (2018)
24. Lano, K.: The UML-RSDS manual. (2020). https://github.com/eclipse/agileuml/blob/master/umlrsds19.pdf
25. Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S.: Declarative specification of bidirectional transformations using design patterns. IEEE Access 7(1), 5222–5249 (2019)
26. Lano, K.: Using the QVT-R analyser and code generator (2020).
27. Lano, K.: QVT2UMLRSDS dataset. (2020). https://doi.org/10.5281/zenodo.3951061
28. Macedo, N., Cunha, A.: Least-change bidirectional model transformation with QVT-R and ATL. SoSyM 15, 783–810 (2016)
29. OMG. Object Constraint Language 2.4 Specification (2014)
30. OMG. MOF Query/View/Transformation v1.3 (2016)
31. OMG, MOF Query/View/Transformation – Open issues (Dec 2019). https://issues.omg.org/issues/lists/qvt-rtf
32. Romeikat, R., Roser, S., Mullender, P., Bauer, B.: Translation of QVT Relations into QVT Operational Mappings, ICMT (2008)
33. Samimi-Dehkordi, L., Zamani, B., Kolahdouz-Rahimi, S.: EvL+Strace: a novel bidirectional model transformation approach. Inf. Softw. Technol. 100, 47–72 (2018)
34. Stevens, P.: Bidirectional MT in QVT: semantic issues and open questions. Sosym 9, 7–20 (2010)
35. Troya, J., Vallecillo, A.: A rewriting logic semantics for ATL. J. Object Technol. 10(5), 1–29 (2011)
36. Westfechtel, B.: Case-based exploration of bidirectional transformations in QVT Relations. SoSyM 17, 989–1029 (2018)
37. Westfechtel, B.: Incremental bidirectional transformations: applying QVT Relations to the Families to Persons benchmark. ENASE 39–53 (2018)
38. Westfechtel, B., Buchmann, T.: Incremental bidirectional transformations: comparing declarative and procedural approaches using the Families to Persons benchmark, ENASE 2018. CCIS 1023, 98–118 (2019)
39. Willink, E.: The micromapping model of computation, ICMT (2017)
40. Willink, E.: MMT/QVT Declarative (QVTd), https://wiki.eclipse.org/MMT/QVT_Declarative_(QVTd) (2019)
41. Willink, E.: QVTd In-place and Copy Transformations (2019). https://wiki.eclipse.org/QVTd_In-place_and_Copy_Transformations
Publisher’s Note  Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

K. Lano has worked for over 25 years in the fields of system specification and verification. He was one of the originators of Model-Driven Engineering and has been a leading advocate of improving the precision of software modelling and in applying software engineering principles to transformation construction. He is the principal developer of the AgileUML toolset which supports the integration of model-based development and agile development.

S. Kolahdouz-Rahimi is an Assistant Professor in the Software Engineering Department at the University of Isfahan. She is an active member of the Model Driven Software Engineering Research Group (MDSE) at this University. She has completed her Ph.D. in Computer Science at Kings College London in 2013. Her current research interests include model-driven software engineering and domain-specific languages.