Software and Systems Modeling (2021) 20:725–766
https://doi.org/10.1007/s10270-020-00824-3

REGULAR PAPER

Implementing QVT-R via semantic interpretation in UML-RSDS

K. Lano[1,2] · **S. Kolahdouz-Rahimi**[1,2]

Received: 13 January 2020 / Revised: 20 July 2020 / Accepted: 11 August 2020 / Published online: 24 September 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 $\tau^{\rightarrow}$ and reverse $\tau^{\leftarrow}$ 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-

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s10270-020-00824-3) contains supplementary material, which is available to authorized users.

✉ K. Lano
kevin.lano@kcl.ac.uk

1 King's College London, London, UK

2 University of Isfahan, Isfahan, Iran

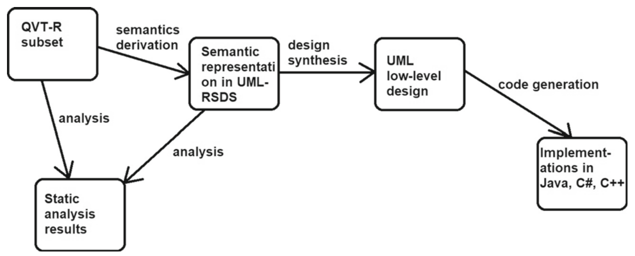


Fig. 1 Analysis and implementation of QVT-R via UML-RSDS

rent 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 QVT2UMLRSDS 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].

Section 2 gives an overview of QVT-R. Section 3 highlights some of the issues which remain unresolved regarding the language semantics and implementation. Section 4 gives an overview of UML-RSDS. Section 5 defines the translation from QVT-R to UML-RSDS for separate-models transformations. Section 6 describes semantic analysis techniques for the translated transformations. Section 7 considers the use of design patterns in QVT-R and UML-RSDS. Section 8 gives an evaluation of the approach by applying it to a number of bx benchmarks from [36]. Section 9 compares our approach to other related work.

In the appendix, "Appendix A" defines the mapping from QVT-R to UML-RSDS for update-in-place transformations. "Appendix B" defines the logical interpretation of QVT-R domains. "Appendix C" gives the definitions of read and write frames of predicates, and of their procedural interpretations. "Appendix D" gives the interpretation of relation overriding and transformation extension. "Appendix E" (in supplementary material) gives the detailed evaluation results on 10 case studies.

2 QVT-R

QVT-R is one of the three MT languages defined in the QVT standard [30], the others being QVT Core (QVT-C) and QVT Operational (QVT-O). QVT-R is intended to enable transformation developers to write high-level and declarative transformation specifications, including bidirectional (bx) and multidirectional transformations, supporting both batch and incremental-mode execution. QVT-O is a unidirectional language oriented towards an imperative style, whilst QVT-C is mainly used as a low-level target language into which QVT-R or other transformation languages can be translated.

2.1 QVT-R transformation structure

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:

$$rule{\to}isUnique(name)$$
$$helpers{\to}isUnique(name)$$
$$rule.name{\to}intersection(helpers.name) = Set\{\}$$

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

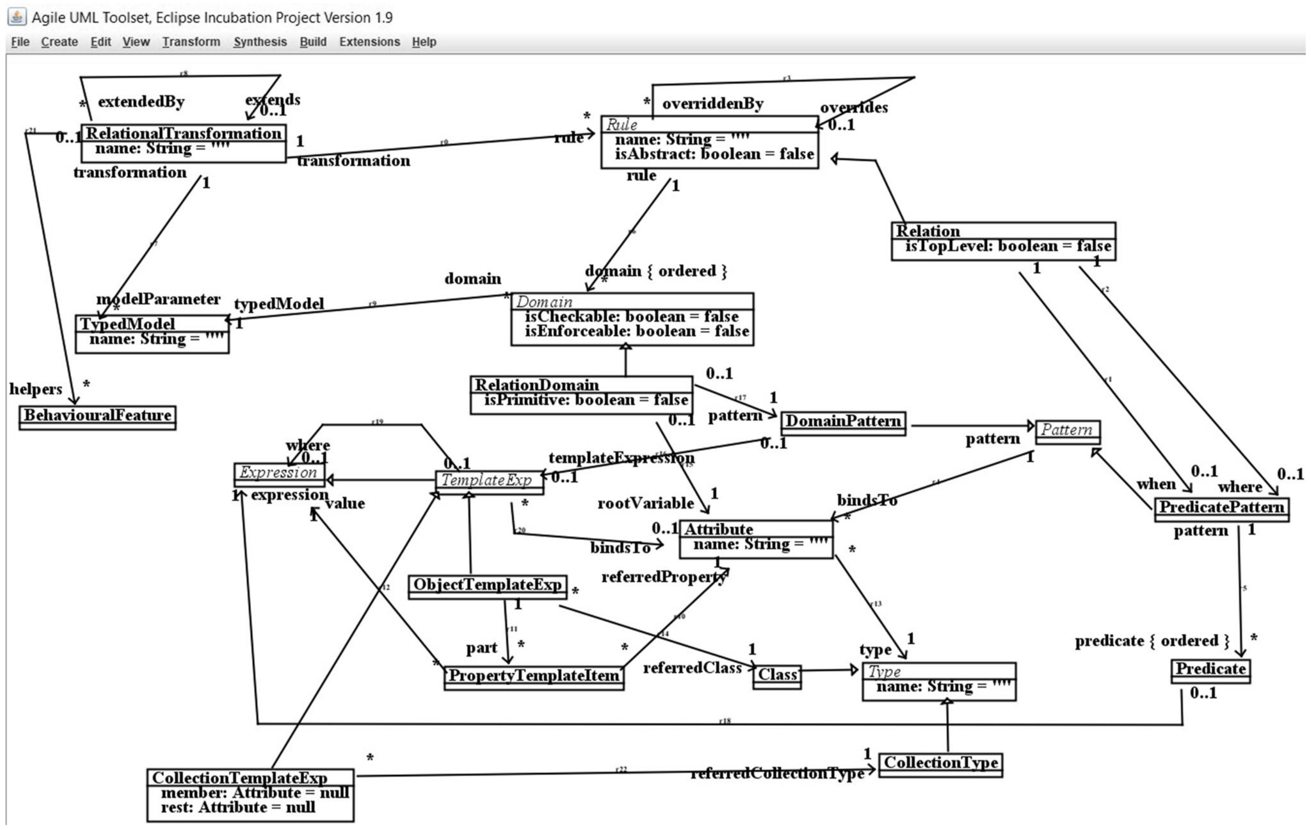


Fig. 2 Subset of QVT-R supported by QVT2UMLRSDS

maps a UML model instance to a C program instance (and vice versa) could be written as a top relation

```
top relation Model2Program
{ enforce domain design u : UMLModel
    { name = n };
 enforce domain 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* (*Relation-Domain::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, $ObjectTemplateExp$s, or collections of source elements, $CollectionTemplateExp$s. 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 variables 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:

```
top relation Model2Program
{ n : String;
 enforce domain design u : UMLModel
    { name = n };
 enforce domain 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

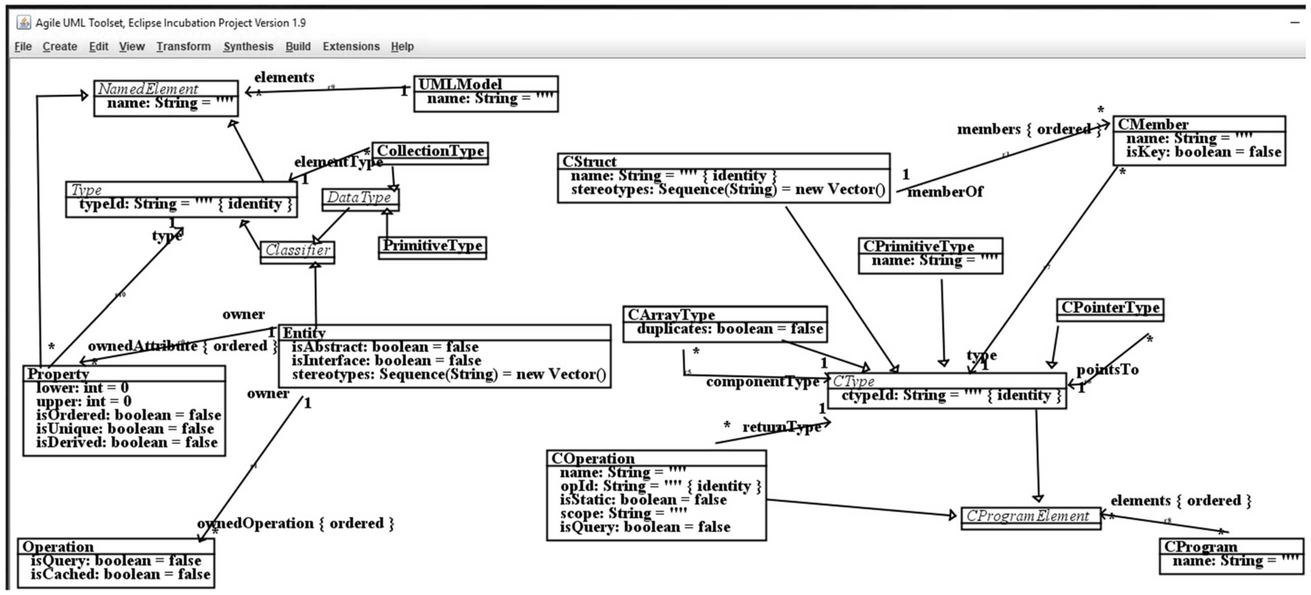

Agile UML Toolset, Eclipse Incubation Project Version 1.9
File Create Edit View Transform Synthesis Build Extensions Help
NamedElement
name: String = ""
UMLModel
name: String = ""
elements
CollectionType
elementType
Type
typeId: String = "" { identity }
DataType
Classifier
PrimitiveType
type
Entity
isAbstract: boolean = false
isInterface: boolean = false
stereotypes: Sequence(String) = new Vector()
owner
ownedAttribute { ordered }
Property
lower: int = 0
upper: int = 0
isOrdered: boolean = false
isUnique: boolean = false
isDerived: boolean = false
ownedOperation { ordered }
Operation
isQuery: boolean = false
isCached: boolean = false
CStruct
name: String = "" { identity }
stereotypes: Sequence(String) = new Vector()
members { ordered }
memberOf
CMember
name: String = ""
isKey: boolean = false
CPrimitiveType
name: String = ""
CArrayType
duplicates: boolean = false
CPointerType
componentType
pointsTo
CType
ctypeId: String = "" { identity }
returnType
COperation
name: String = ""
opId: String = "" { identity }
isStatic: boolean = false
scope: String = ""
isQuery: boolean = false
CProgramElement
elements { ordered }
CProgram
name: String = ""

Fig. 3 Simplified UML design and C metamodels

models. Elements within the target model can also be created or deleted.

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

An important mechanism in QVT-R is *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 $Model2Program$ in the C direction, with a non-empty C model, the creation of a new $CProgram$ only occurs if there is not already a $CProgram$ instance satisfying the required property $name = u.name$, 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:

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

The $typeId$ and $ctypeId$ 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 $e : Entity$, both a $CPointerType$ instance and a linked $CStruct$ instance are potentially created. In general, any number of linked elements may be defined in a domain template. We describe this situation as (vertical) $entity\ splitting$ [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

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

and

$$p : CPointerType\ and\ p.ctypeId = t\ and\ c : CStruct\ and\\ p.pointsTo = c\ and\ c.name = n\ and\ c.ctypeId = n$$

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

```
top relation Class2CPointerTypeV2
{ enforce domain design e : Entity
   { typeId = t, name = n }
     { e.isAbstract = false };
 enforce domain C p : CPointerType
   { ctypeId = t, pointsTo = c : CStruct
      { name = n, ctypeId = n } };
}
```

In the C direction, this additional condition restricts the mapping to concrete UML classes. In the design direction, the condition assigns $e.isAbstract = false$ for the target $e : Entity$.

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 $when\ \{P(y)\}$ 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:

```
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 $s : CStruct$ must already have been created for $e = p.owner$, before $Property2CMember$ can be applied to create $m : CMember$ for p. The relation $Class2CStruct$ is:

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

Using this relation, $Class2CPointerType$ can be respecified as:

```
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

```
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:

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

The semantics of $overrides$ 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 $S1$ or $T1$ 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 $name : String$) or 1-multiplicity reference features (such as $owner : Entity$) 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:

```
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 $e.ownedAttribute$ is empty, then no $p : Property$ 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, 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 $e.ownedAttribute \rightarrow forAll(p|...)$ 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 concrete top 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 template 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 nondeterministic 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 of the target [37]. In some cases, check-before-enforce is the incorrect semantics for a transformation, for example, in cases where there are no keys, but nonetheless a 1–1 mapping of source elements to target elements is required, as in the Families to Persons case [37]. A more subtle case is where intermediate objects in a chain of linked objects should not be overwritten (Sect. 6).

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:

```
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 *where* clause and target domains, nor 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 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 dag2ast/ast2dag 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 bag22bag1 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 ecore2sql3 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 τ which have preconditions Pre_τ and a sequence of postcondition constraints $Post_\tau$ 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 $s \rightarrow includes(x)$[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 $\ll extend \gg$ or $\ll include \gg$ 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

$$\begin{aligned} &E :: \\ &\quad Pre \implies Post \end{aligned}$$

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 $stat_{LC}(Post)$ as a UML activity[2]. The context class E is optional. As in UML operation postconditions, prestate forms v@pre of variables and expressions v can be used, to refer to the read-only state of v at initiation 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 $uc_1, \ldots, uc_n$. The activity of uc is defined as the sequential composition of the activities of the uc_i.

4.2 UML-RSDS rule semantics

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

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

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

OCL predicates such as $p.name = name$ and $E \rightarrow exists(e|P)$ 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 $stat(P)$ 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 semantics of P. An alternative "least-change" semantics is given by an activity $stat_{LC}(P)$ [25]. This tries to minimise changes to $wr(P)$ data and to maximise reuse of existing elements. Definitions of rd, wr, $stat$ and $stat_{LC}$ are given in "Appendix C". For example, the $UMLModel$ rule has read frame $\{UMLModel, UMLModel::name\}$ and write frame $\{CProgram, CProgram::name\}$. The "design synthesis" step of Fig. 1 involves the derivation of the $stat(Post_\tau)$ or $stat_{LC}(Post_\tau)$ activities for each use case τ (a $\ll bx \gg$ stereotype on a use case uc indicates that $stat_{LC}$ should be used for uc).

The QVT-R check-before-enforce mechanism for such a rule could be expressed as

$$E :: \\ \quad Pre \; \& \; not(Post@pre) \implies Post$$

In other words, execution of the succedent 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 entire postcondition, we can use the $stat_{LC}(Post)$ activity, which uses local checks to avoid un-necessary updates. For example, $stat_{LC}(x : y.r)$ for *-multiplicity r first tests if $y.r{\rightarrow}includes(x)$ 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, $Class2CPointerType$ could be written as:

```
Entity::
 CPointerType->exists( p | p.ctypeId
   = typeId &
   CStruct->exists( c | c.ctypeId
   = name & c.name = name
     & p.pointsTo = c ) )
```

UML-RSDS specifications usually make use of identity attributes (unique keys) to look up previously created elements. This acts as a persistent trace mechanism. For example, $typeId$ and $ctypeId$ are identity attributes, so $CType$ instances can be looked up by their $ctypeId$ values: $CType[val]$ denotes the unique $CType$ instance t with $t.ctypeId = val$, if any such instance exists. In the $CStruct{\rightarrow}exists$ quantifier above, if the element $CStruct[name]$ already exists, then it is bound to c and the statements $stat(c.name = name)$ and $stat(p.pointsTo = c)$ are executed.

Using this mechanism, the equivalent of the $Property2CMember$ rule is:

```
Property::
 CMember->exists( m | m.memberOf
    = CPointerType[owner.typeId].pointsTo &
    m.name = name & m.isKey = isUnique )
```

where it is assumed that the $CPointerType$ instance with $ctypeId = owner.typeId$ already exists. In UML-RSDS, setting one end of a bidirectional association such as $members/memberOf$ implicitly also updates the other end (here, adding m to the members of the selected $CStruct$). 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 $E[val]$ also applies to collections of key values: $CType[vals]$ for collection $vals$ is the collection of instances of $CType$ with $ctypeId$ value in $vals$.

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

```
Property::
 p = CPointerType[owner.typeId] =>
   CMember->exists( m |
         m.memberOf = p.pointsTo & m.name
        = name & m.isKey = isUnique )
```

Here, p is an auxiliary variable, similar to an OCL let variable or QVT-R local variable. It is implicitly ∀-quantified over the entire constraint.

There are two versions of the general existential quantifier. ${\rightarrow}exists_{LC}$ 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 $key : String$ identity attribute, the $stat$ or $stat_{LC}$ design of this quantifier expression uses any key specification $x.key = value$ 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 $stat(Q)$ or $stat_{LC}(Q)$. If E has no identity attribute key, then with $stat$ semantics a new x is always created and P is established for this x using $stat(P)$.
2. $E{\rightarrow}existsLC(x|P)$—in the case that E has a $key : String$ identity attribute, the $stat$ or $stat_{LC}$ design of this quantifier expression uses any key specification $x.key = value$ 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 $stat_{LC}(Q)$. 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 $stat_{LC}$. 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 $existsLC$ are equivalent, and test that $E \rightarrow select(x|P)$ is non-empty.

4.3 UML-RSDS transformation semantics

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

```
CProgram::
 UMLModel->exists( u | u.name = name )
```

This syntactic inversion is based on the semantic concept of a rule invariant. The conjunction of the inverted rules forms a transformation invariant Inv_τ of τ [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 τ, 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 $Entity$ above must precede the rule for $Property$, because the $Entity$ rule writes $CPointerType$, which is read by the $Property$ rule. This condition is termed *syntactic non-interference*. The activity synthesised for such a τ by design synthesis is $stat(r_1); \ldots; stat(r_n)$.

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 $E :: \quad P$ to be used, rather than a fixed-point implementation (which would repeatedly apply $stat(P)$ 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:

$$x \rightarrow isDeleted()$$

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 $Post_\tau$ constraints of a use case τ are ordered so that no constraint r_i can be invalidated by later rules r_j:

$$r_i \implies [stat(r_j)]r_i$$

for $i < j$, then the semantic effect of the UML-RSDS use case τ 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 Inv_τ 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 $stat_{LC}$
Single and compound keys for elements	Single keys
Query operations	Query and update operations, including cached query operations
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 transformation composition	Internal transformation decomposition 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
when clause	Expressions including trace tests
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 $P(sv, tv)$ on source and target variables sv, tv, with respect to a binding $sv \mapsto s$ of source elements s to sv:

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 $P(sv, tv)$ with the binding $tv \mapsto t$.

Table 3 Element mappings and pattern usage in QVT-R cases

Transformation	Size	Element mappings	Patterns/techniques
AbstractToConcrete	47	Update-in-place	
hsm2nhsm (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 1–1 using keys	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	
ast2dag	439	$n-1$ using keys	Marker relation, Map objects before links
f2p/p2f	462	1–1 using procedural Coding	
Bpmn2UseCase	532	$1-n$	(vertical) Entity splitting
Communication2class	1029	1–1 or $n-1$ Using keys	Explicit checks to avoid duplicate target elements
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%
1–1 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

(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 $stat_{LC}$ 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 τ 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 τ refers to a target element t, then any target element x from which deletion could propagate to t must also be referenced by τ.
- (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 τ 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 τ. (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 τ is semantically represented as a UML-RSDS use case τ'. τ 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 τ satisfies the semantic correctness properties of (a) to (e) of Sect. 5.4.

Key declarations $key\ E\ \{p\}$ of τ 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

$$
\begin{aligned}
&E \rightarrow forAll(e_1 | \\
&\quad E \rightarrow forAll(e_2 | e_1.p_1 = e_2.p_1\ \&\ \ldots\ \&\ e_1. \\
&p_n = e_2.p_n \implies e_1 = e_2))
\end{aligned}
$$

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

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

As in the QVT-R to QVT-Core mapping of [30], we use trace classes $R\$trace$ 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 $R\$trace$, 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,

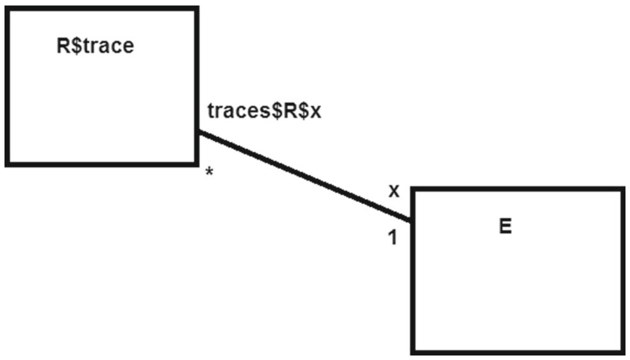


Fig. 4 Trace class definition

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 $\bigcup_{d:sdom} ovars_d$ of its source (non-target) domains $sdom$, and the *target object variables* $tvars_R$ are the object variables $\bigcup_{d:tdom} ovars_d$ 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, $R\$trace$ 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*

Table 5 Variables associated with relations

Notation	Meaning
$sdom$	The source domains $d \in R.domain$, including primitive domains
$tdom$	The target domains d of $R.domain$
$ovars_d$	Object variables of domain d, including the root variable
$svars_R$	$\bigcup_{d \in sdom} ovars_d$
$tvars_R$	$\bigcup_{d \in tdom} ovars_d$
$ovars_R$	$svars_R \cup tvars_R$
$sourcevars_R$	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 $tdom$ $sourcevars_R \subseteq avars_R$
$whenvars_R$	Variables free in $when$ clause of R $whenvars_R \subseteq avars_R$
$invars_R$	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

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 $tr : R\$trace$ 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 $tr : R\$trace \ \& \ a_1 = tr.v_1 \ \& \ \dots \ \& \ a_n = tr.v_n$ 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 $not(R(a_1, ..., a_n))$ is interpreted as $not(R\$trace \rightarrow exists(tr | tr.v_1 = a_1 \ \& \dots \& \ tr.v_n =$

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 $outvars_R$ 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 $v \in ovars_R$ for a successful application of R. Equivalently, x occurs in some $R\$trace$ tuple
Elements referenced by a transformation execution	Elements referenced by any rule application in the transformation execution
Application condition of relation R	Condition on $svars_R$ determining if R can be applied to particular source model elements
Effective for update in direction m	An expression e that has a defined $stat(e)$, and with $wr(e)$ a non-empty subset of the variables ($targetvars_R$) 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 $tr \in R\$trace$ linked to these elements (the same approach is used in the translation of [8] from QVT-R to CPN). However, membership of $R\$trace$ 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 $R\$trace$-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* $Pres_R(m)$ expresses the effect of R when applied to tuples $elems$ of source and target elements which are linked to one $R\$trace$ 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 $Pres_R(m)$ is to modify target model data linked to the trace, to re-establish the R target domain properties for $elems$ relative to the source data. Traces which cannot be repaired are deleted.
- *Construction constraints* $Con_R(m)$ expresses the effect of R when applied to source elements/tuples $selems$ which are not in any $R\$trace$, and therefore have not yet had R applied to them. $Con_R(m)$ 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 $selems$ and their matched target elements.
- *Cleanup constraints* $Cleanup_E(m)$ manages the deletion of target instances $e : E$ that are not linked to any source instance via any valid trace $R\$trace$ for any relation R that may create E instances.

The constraints are grouped into three phases $Pres_\tau(m)$, $Con_\tau(m)$ and $Cleanup_\tau(m)$ 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 $when$ clause are tested together, and if these are satisfiable, then the target domains are executed, followed by the $where$ clause. Execution order within target domains and the $where$ clause follows textual ordering (left to right and top to bottom).

In the semantic representation, this is expressed by defining $Pres_R(m)$ and $Con_R(m)$ via an OCL formula $\theta_R(m, vars)$ of the form

$$cpreds(sdom, vars) \ \& \ whenp \implies epreds(tdom, vars \cup whenvars_R \cup sourcevars_R) \ \& \ wherep$$

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

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

$whenp$ is the logical interpretation of the $when$ clause: the conjunction of the clause predicates, with relation calls treated as tests on the relation trace.

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

$cpreds(sdom, bound)$ is the logical interpretation of the non-target domains $sdom$, given a set of currently bound variables $bound$, and $epreds(tdom, bound)$ is the interpretation of the target domains $tdom$, i.e. domains d with model m. The scope of any $exists$ or $existsLC$ quantifiers introduced for target object variables in the $epreds$ formulae is extended over the remainder of the succedent, including $wherep$ (as in [30], Annex B). We define $cpreds$ and $epreds$ in "Appendix B".

We write $\theta_R(m)$ for $\theta_R(m, ovars_R)$. A predicate $guard_R(m)$ is formed as the antecedent $\varphi_R(m)$ of $\theta_R(m)$, with all variables apart from the $ovars_R$ variables ∃-quantified. That is, as an OCL formula it is

$$T_1 \rightarrow exists(v_1 | ... T_k \rightarrow exists(v_k | \varphi_R(m))...)$$

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

The *checkonly* semantics of R in the direction of m asserts that at termination of τ, for every tuple of $svars_R$ elements that satisfy $sguard_R(m)$, there is an extended tuple of elements for $ovars_R$ which satisfies $\theta_R(m)$ (Section 7.10.1 of [30]).

Regarding the $overrides$ 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 $when$ test on an overridden abstract relation $R(pars)$ is replaced by a disjunction $R_1(pars) \ or \ ... \ or \ R_n(pars)$ testing the concrete relations R_i that override R. This then becomes a test on their trace relations in the semantics. A $where$ call of a non-top abstract $R(pars)$ is semantically expressed as a conditional

```
$if\ guard_{R_1}[pars/ovars_{R_1}]\ then\ Op_{R_1}(pars)$
$else\ ...$
$else\ if\ guard_{R_n}[pars/ovars_{R_n}]\ then\ Op_{R_n}(pars)$
$endif\ ...\ endif$
```

The 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 $extends$ mechanism ("Appendix C"). Thus, in the following we will only consider concrete relations and transformations without $extends$.

5.2.1 Preservation constraints

The first execution phase $Pres_\tau(m)$ of τ' applies for incremental-mode execution only, using a persistent trace. For each relation R, $Pres_R(m)$ 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 τ); it is defined schematically as

```
$R\$trace@pre$ ::
    $if\ guard_R(m)\ then\ \theta_R(m)$
    $else\ self \rightarrow isDeleted()$
    $endif$
```

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 $when$ 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 $when$ or $where$ clause (the conditions (a) and (e) of Sect. 5.4), then $Pres_R(m)$ has disjoint read and write frames. In addition, elements of $R\$trace$ are only deleted, not created. This means that a polynomial-time complexity bounded-loop implementation of $Pres_R$ is sufficient, and we enforce this design choice by using the @pre annotation on $R\$trace$.

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

$Pres_\tau(m)$ propagates element feature changes from source models to m. For example, $Pres_{Model2Program}(C)$ is

```
Model2Program$trace@pre::
 if u : UMLModel
 then (n = u.name => p.name = n)
 else self->isDeleted()
 endif
```

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

5.2.2 Construction constraints

The second phase $Con_\tau(m)$ of τ' 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:

$$
\begin{array}{l}
:: \\
\quad cpreds(sdom, \{\}) \mathbin{\&} whenp \mathbin{\&} \\
\quad not(R\$trace@pre \rightarrow exists(tr | \&_{sv \in svars_R} tr.sv = sv)) \implies \\
\qquad epreds(tdom, whenvars_R \cup sourcevars_R) \mathbin{\&} wherepx
\end{array}
$$

$wherepx$ is $wherep$ right conjoined with the additional creation $R\$trace \rightarrow exists(tr | \&_{sv \in svars_R} tr.sv = sv \mathbin{\&} \&_{tv \in tvars_R} tr.tv = tv)$ of a new trace instance.

We use $R\$trace$@pre in 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 $exists$ or $existsLC$ quantifiers introduced by $epreds(...)$ in the succedent apply over all of the succedent following their introduction, by definition of $epreds$. 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, $Con_{Model2Program}(C)$ is:

```
::
 u : UMLModel & n = u.name &
 not(Model2Program$trace@pre->exists
    ( tr | tr.u = u ))   =>
       CProgram->exists( p | p.name = n   &
           Model2Program$trace->exists
         ( tr | tr.u = u & tr.p = p ) )
```

5.2.3 Cleanup constraints

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

$$
\begin{array}{l}
E :: \\
\quad not(R1\$trace \rightarrow exists(r1x | r1x.e1 = self)) \\
\ \mathbin{\&} \dots \mathbin{\&} \\
\quad not(Rk\$trace \rightarrow exists(rkx | rkx.em = self)) \\
\ \implies self \rightarrow isDeleted()
\end{array}
$$

where the Ri, $i = 1$ to k, are all the relations (top or non-top) in which the entity E occurs as the type of some target object variable $ei : E$ in $outvars_{Ri}$. All $Cleanup_E(m)$ constraints for entity types of m are placed in a final τ' 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 $CProgram$ instances, $Model2Program$, then $Cleanup_{CProgram}(C)$ is:

```
CProgram::
 not(Model2Program$trace->exists
      ( tr | tr.p = self)) => 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:

$$
\begin{array}{l}
E :: \\
\quad not(R1\$trace \rightarrow exists(r1x | r1x.e1 = self \mathbin{\&} r1x.guard_{R1}(m))) \mathbin{\&} \dots \mathbin{\&} \\
\quad not(Rk\$trace \rightarrow exists(rkx | rkx.em = self \mathbin{\&} rkx.guard_{Rk}(m))) \implies self \rightarrow isDeleted()
\end{array}
$$

$rx.P$ is P with variables u of $ovars_R$ in P replaced by $rx.u$. This version of the cleanup constraint deletes any target element which does not occur in any Ri trace that satisfies $guard_{Ri}(m)$. 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 postconditions 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 *cpreds* and *epreds* but including the root variables of the relation domains in the bound variable sets:

$$cpreds(sdom, domain.rootVariable)\ \&\ whenp\ \& \\ not(R\$trace@pre{\rightarrow}exists(tr|\&_{sv \in svars_R} tr.sv = sv\ \&\ \&_{tv \in tvars'_R} tr.tv = tv)) \implies \\ epreds(tdom, whenvars_R \cup domain.rootVariable \cup sourcevars_R)\ \&\ wherepx$$

$tvars'_R$ is $tvars_R \cap domain.rootVariable$. Target objects can be created using non-root object templates in target domains.

For example, the non-top relation $NonTopProperty2CMember$ has the interpretation in the C direction as an update operation

```
NonTopProperty2CMember_C(e : Entity, c :
    CStruct)
post:
 p : e.ownedAttribute & n = p.name
    & k = p.isUnique &
 not(NonTopProperty2CMember$trace@pre->
    exists( tr |
     tr.e = e & tr.p = p & tr.c = c ) )=>
         CMember->exists( m | m :
           c.members &
            m.name = n & m.isKey = k &
            NonTopProperty2CMember$trace->
          exists( tr |
           tr.e = e & tr.p = p & tr.
          c = c & tr.m = m )  )
```

These operations are added as static owned operations of the use case $Con_\tau(m)$ representing the transformation phase.

For $Pres_\tau(m)$, a different version of the operation R_m is defined, with postcondition $PresOp_R(m)$:

$$tr : R\$trace\ \&\ v_1 = tr.v_1\ \&\ ...\ \&\ v_n = tr.v_n \implies \\ if\ guard_R(m) \\ then\ \theta_R(m) \\ else\ tr{\rightarrow}isDeleted() \\ endif$$

The $v_1, \ldots, v_n$ are $ovars_R$. The operation R_m is added as a static owned operation of $Pres_\tau(m)$.

This version of $NonTopProperty2CMember_C$ has the form:

```
NonTopProperty2CMember_C(e : Entity,
   c : CStruct)
post:
 tr : NonTopProperty2CMember$trace
    & e = tr.e & p = tr.p &
 c = tr.c & m = tr.m  =>
   if p : e.ownedAttribute
   then (n = p.name & k = p.isUnique  =>
          m : c.members & m.
   name = n & m.isKey = k)
  else tr->isDeleted()
  endif
```

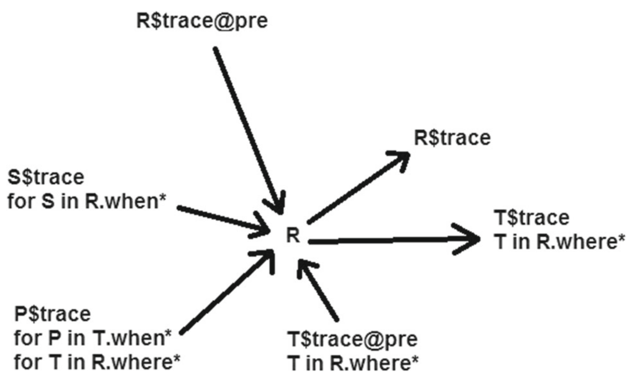


Fig. 5 Trace data dependencies of Con_R, $Pres_R$

5.2.5 Overall transformation semantics

For each execution direction of a separate-models transformation τ, the three phases described above are executed in the order $Pres_\tau(m)$; $Con_\tau(m)$; $Cleanup_\tau(m)$. The semantics generalises in a natural manner to consider execution directed at multiple target models, extending [30].

Figure 5 shows the data dependencies of Con_R and $Pres_R$ wrt trace entities, for the default semantics. $where*$ is the recursive closure of relation calling though $where$ clauses, likewise for $when*$. 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 between the constraints within transformations Con_τ and $Pres_\tau$, quite strict constraints must be placed on how relations depend on each other via $when$ and $where$ clauses. These conditions are encoded in condition (e) of the following section.

$Cleanup_E$ reads and writes E, and reads $R\$trace$ for any relation R with an $outvars_R$ 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 τ (occurring in some trace tuple) cannot be affected by cascaded deletion from target elements that are not referenced by τ (not occurring in any trace tuple).

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:

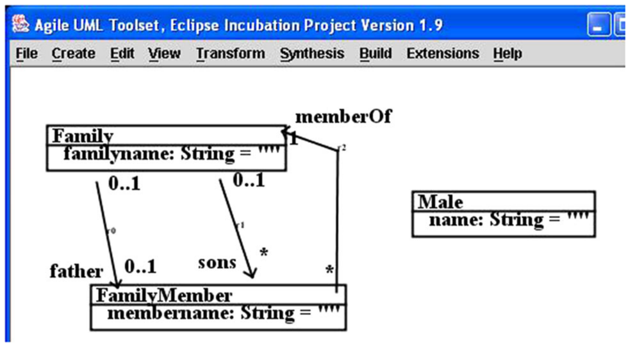

Agile UML Toolset, Eclipse Incubation Project Version 1.9
File Create Edit View Transform Synthesis Build Extensions Help
memberOf
Family
familyname: String = ""
0..1
0..1
1
Male
name: String = ""
father
0..1
sons
*
*
FamilyMember
membername: String = ""

Fig. 6 Simple Families to Persons metamodels

```
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
```

```
::
fm : FamilyMember & n = fm.membername & fm.
    memberOf.father->includes(fm) &
not(fm.traces$FamilyMember2Male$fm@pre->exists
    ( tr$1 | true )) =>
     Male->exists( m | m.name = fm.memberOf.
     familyname + ", " + n &
     FamilyMember2Male$trace->exists( tr$0 |
     tr$0.fm = fm & tr$0.m = m ) )


::
fm : FamilyMember & n = fm.membername & fm.
     memberOf.sons->includes(fm) &
not(fm.traces$FamilyMember2Male$fm@pre->exists
    ( tr$1 | true )) =>
     Male->exists( m | m.name = fm.memberOf.
     familyname + ", " + n &
     FamilyMember2Male$trace->exists( tr$0 |
     tr$0.fm = fm & tr$0.m = m ) )


Use Case, name: tau$Cleanup

Male::
traces$FamilyMember2Male$m@pre->isEmpty() =>
     self->isDeleted()
```

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 $Pres_\tau(m)$, $Con_\tau(m)$, $Cleanup_\tau(m)$ of τ' should establish the following consistency relations Rel_τ 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: $E{\rightarrow}forAll(tx|R1\$trace{\rightarrow}exists(r1|r1.t1 = tx)$ $or \ldots or$ $Rk\$trace{\rightarrow}exists(rk|rk.tk = tx))$ where the Ri are as for $Cleanup_E(m)$.

2. That all instances tr of $R\$trace$ satisfy the logical relation $guard_R(m)$ & $\theta_R(m)$ defined by R, for each relation R: $tr.guard_R(m)$ & $tr.\theta_R(m)$.
3. That any tuple of source elements for $svars_R$ that satisfies $sguard_R(m)$ is linked to some $R\$trace$ instance tr, for top relations R.

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

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 $ovars_R = \{v_i\}_i$ of R:

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

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 $guard_R(m)(vals)$ & $\theta_R(m)(vals)$ but not be in $R\$trace$, because $vals$ has not been processed by R. For example, consider the transformation with top rules:

```
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 $guard_{R1}(trg)[a1/a]$ & $\theta_{R1}(trg)[a1, b2/a, b]$ holds but not $(a1, b2) \in R1\$trace$. $R1$ only selects one possible B instance to match to $a1$, in this case $b1$.

The $\Longrightarrow$ direction can also fail, because conflicts between relations may invalidate existing traces. For example, if $id : String$ 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

```
top relation R1X
{ enforce domain src a : A
   { id = ix, value = v };
 enforce domain trg b : B
   { id = ix, value = v };
}

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

can conflict, with some $b : B$ created by $R1X$ and hence with $b = tr.b$ for some $tr : R1X\$trace$, 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 $when$ test $not(R1(v))$ on another relation. Subsequent execution of $R1$ may invalidate the guard of $R2$ and hence invalidate the $\Longrightarrow$ 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 τ directed at a model m. These conditions ensure that the postconditions (1), (2), (3) are established by the semantic interpretation τ' of τ:

- (a) "No secretly created objects": No target features/entity names of TL occur in the $when$ 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 $where$ clause should not conflict with target domain templates;
- (d) "No secretly deleted objects": If target element $t : A$ is referenced by τ, 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

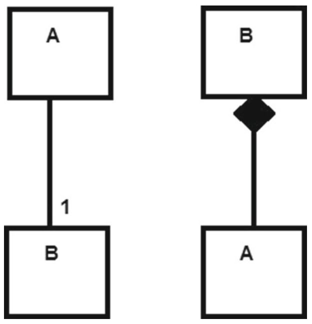


Fig. 7 Deletion propagation (from B to A) situations where condition (d) applies

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 τ 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 $when$ 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 $when$ or $where$ clauses[3]. All non-top relations are called from at least one top relation directly or indirectly. No relation can be called via both $when$ and $where$ 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. $R_i\ called\ in\ R_j.when \implies i < j$
  2. $S\ called\ in\ R_j.where*$ and $R_i\ called\ in\ S.when* \implies i < j$
  3. $S\ called\ in\ R_i.where*$ and $S\ called\ in\ R_j.when* \implies i < j$.

Condition (a) prevents relations from referring to arbitrary target elements or to the extents $T.allInstances()$ of target classes T. Thus, predicates such as $v = T.allInstances() \rightarrow size()$ or $T \rightarrow exists(t|P)$ 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 $Pres_\tau$ and Con_τ 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 Con_τ and $Pres_\tau$ 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, $closure$ expressions [28] or $forAll$ quantifiers.

Theorem 1 *If the correctness conditions (a) to (e) hold for a separate-models transformation τ, then a partial execution order $\ll$ 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, $\ll$ is defined as $<$. For non-top R, top P, $R \ll P$ means that any top S with $R \in S.where*$ must have $S < P$. For non-top P, top R, $R \ll P$ means that any top S with $P \in S.where*$ must have $R < S$. For non-top P, R, $R \ll P$ means that any top S, T with $R \in S.where*$, $P \in T.where*$ must 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 $R \in S.where*$ 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 $P \in S.where*$, $R \in T.where*$ has $S < T$. But $R \ll P$ would imply $T < S$.

Thus, provided that every non-top relation P has some top R with $P \in R.where*$, the result follows. □

This suggests a way to organise QVT-R transformations: list the top relations in $<$ order, and group the non-top relations of $R.where*$ for top relation R together with R.

Note that it is possible for τ to satisfy the correctness conditions (a) to (e) in one execution direction but not in others. For a 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 $when$ clauses of 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 τ, the correctness conditions (a) to (e) ensure that τ establishes or preserves the properties (1), (2) and (3).*

[3] Self-recursion via $where$ is permitted for non-top relations by our translation, but the specifier must then ensure this does not result in conflicting updates or non-termination.

Proof For batch-mode execution, assuming (a), (b), (c), (d), (e), the $Con_\tau(m)$ phase establishes invariant (3), since if tuple v satisfies $sguard_R(m)$ for concrete top relation R at termination of τ, i.e. $sguard_R(m)[v/svars_R]$, it also did so at the start of the $Con_R(m)$ execution step of $Con_\tau(m)$—because $sguard_R(m)$ 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 $sguard_R$, 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 $Cleanup_\tau(m)$ (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_P applications in $Con_\tau(m)$, either of R or another rule, by conditions (b), (c), (e). Condition (e) also ensures termination of the Con_τ phase.

The $Cleanup_\tau(m)$ 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 $Cleanup_\tau$. For separate-models transformations, the $Cleanup_\tau(m)$ 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 $Pres_\tau(m)$, $Con_\tau(m)$ or $Cleanup_\tau(m)$ have any effect. (2) means that the $Pres_\tau(m)$ 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 $Cleanup_\tau(m)$.

Assuming the above conditions (a) to (e) for transformation τ, $Pres_\tau(m)$ 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. $Con_\tau(m)$ 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, $Cleanup_\tau(m)$ 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 $Con_\tau(m)$ use case is necessary.

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

Proof Theorem 2 shows that the conditions (1), (2), (3) are established by τ' at its termination. For each concrete top relation R of τ, conditions (2) and (3) imply that for any source element tuple x for $svars_R$ which satisfies $sguard_R(m)$, there is an extended tuple y of elements for $ovars_R$ 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 $existsLC$ 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 $wherepx$, 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 updator 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 $getVarsOfExp$ function, and the $RExpToMExp$ and $TopLevelRelationToMappingForEnforcement$ 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 $sguard_R(m)$ of a relation R with source domain $s : SST$ for SST equal to ST or a supertype of ST, then sx will be processed by $Con_R(m)$ 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 $s : ST$*—if s occurs in some trace $R\$trace$, and $guard_R(m)$ remains true for s, then $Pres_R(m)$ applies to update linked target elements appropriately. If s is in some $tr : R\$trace$ but $guard_R(m)$ is falsified by the change of f/att value, then tr is removed from $R\$trace$ by $Pres_\tau(m)$, and $Cleanup_\tau(m)$ removes any target elements that depended exclusively on s. If s does not occur in any trace, but the change in f/att now validates some $guard_R$, then $Con_R(m)$ is applied to s.
- *Addition of an element to a collection-valued feature f of existing element $s : ST$*—if s is in $tr : R\$trace$ and $guard_R(m)$ remains true, then $Pres_R(m)$ propagates the change to corresponding target elements. The cases of $guard_R(m)$ being falsified and s not in $R\$trace$ 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 Con_R (or $ConOp_R$ for non-top R). Two cases need to be distinguished:

1. Either there is already a tuple of elements of $outvars_R$ which satisfy the required constraints

   $$epreds(tdom, whenvars_R \cup sourcevars_R \cup outvars_R)\ \&\ wherep$$

   In this case, the elements are selected and a new $R\$trace$ instance created for $ovars_R$, or
2. No such tuple exists, and new target elements need to be created and updated using $epreds(tdom, whenvars_R \cup sourcevars_R)\ \&\ wherep$, and the resulting $ovars_R$ elements linked in a new trace element.

In order to achieve this logic, we need an extended OCL let operator, which generalises the version defined in [29]. This has the form

$$let\ vars\ be\ P\ in\ Q$$

where $vars$ are a list of variables, P are constraints on the $vars$, and Q is a constraint which has a $stat(Q)$ interpretation and does not write to the variables of P. This generalised let has a procedural interpretation as an 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:

```
::
    $cpreds(sdom, \{\})$ & $whenp$ &
    $not(R\$trace@pre \rightarrow exists(tr | \&_{sv \in svars_R} tr.sv = sv))$ $\implies$
            $if\ \exists v \cdot (epreds(tdom, whenvars_R \cup sourcevars_R \cup outvars_R)\ \&\ wherep)[v/outvars_R]@pre$
            $then$
                    $let\ outvars_R$
                    $be\ epreds(tdom, whenvars_R \cup sourcevars_R \cup outvars_R)\ \&\ wherep$
                    $in\ CreateTrace_R$
            $else$
                    $epreds(tdom, whenvars_R \cup sourcevars_R)\ \&\ wherepx$
            $endif$
```

where $CreateTrace_R$ is $R\$trace \rightarrow exists(tr | \&_{sv \in svars_R} tr.sv = sv\ \&\ \&_{tv \in tvars_R} tr.tv = tv)$

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 { pattern }
```

for target object templates where least-change partial matching is required. Such templates have the semantic interpretation $E \rightarrow existsLC(e|Pred)$ in the $epreds$ predicate, instead of $E \rightarrow exists(e|Pred)$, where $Pred$ semantically interprets $pattern$. 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 τ' of τ on the basis that trace instances of the $R\$trace$ classes are persisted to support incremental execution. However, phases $Con_\tau(m)$ and $Cleanup_\tau(m)$ are also applicable in the case of non-persistent traces. The properties (1), (2), (3) remain valid as postcondition properties established at termination of τ', and the proof of Theorem 2 remains valid for batch-mode execution.

However, incremental execution mode is no longer supported. Execution of τ' 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 $Pres_\tau(m)$ is not available. Each $Con_R(m)$ 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 $undo_R(m)$ predicate can be derived, which is a conjunction of predicates $y.r \rightarrow excludes(x)$ for each predicate $y.r \rightarrow includes(x)$ or $x : y.r$ in $epreds(tdom, ovars_R \cup whenvars_R \cup sourcevars_R)\ \&\ wherep$, where x and y are in $ovars_R$. $Pres_R(m)$ is then modified to perform $undo_R(m)$ if the guard of R is false for an existing trace:

```
$R\$trace@pre$ ::
    $if\ guard_R(m)\ then\ \theta_R(m)$
    $else\ undo_R(m)\ \&\ self \rightarrow isDeleted()$
    $endif$
```

Similarly for $PresOp_R$ 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
 };

 ...

 subtransformation tn(parsn)
 { relations for tn,
   operations used only in tn
 };
}
```

Each of the subtransformation parameter lists $pars_i$ is a subsequence of the main transformation parameters $pars$.

A sequential composition $\tau = \tau_1; \tau_2$ of two QVT-R transformations can be given a semantics as a sequential composition of the corresponding UML-RSDS transformations $\tau_1'; \tau_2'$. This chains the three phases of τ_2' after the three phases of τ_1'. Provided that the write frames of τ_1 and τ_2 are disjoint, and that τ_2 does not write to any data read by τ_1, then the composed effect of τ' is well defined: τ_2' does not invalidate the postconditions established by τ_1' on its models.

6 Semantic analysis of QVT-R

The translation of a QVT-R relation R to an OCL constraint $\theta_R(m)$ immediately exposes the semantic form of the relation in terms of quantified source and target elements.

In general, each relation should only concern a single one-to-many association within each model [16]. Hence, each relation should refer to elements at two composition levels or fewer, per model (e.g. to programs and classes, or to classes and attributes, but not to programs, classes and attributes). Multiple levels of quantification within one relation are both difficult to comprehend (cf. the AbstractToConcrete case from Modelmorf) and inefficient to execute.

The problems identified with the $Class2CStructV2$ relation in Sect. 2 can now be understood by considering the form of the semantics $\theta_{Class2CStructV2}(C)$ of the relation in UML-RSDS notation:

```
e : Entity & n = e.name & p :
    e.ownedAttribute &
n1 = p.name & k = p.isUnique =>
   CStruct->exists( c |
       c.name = n & c.ctypeId = n &
       CMember->exists( m | m :
    c.members & m.name = n1
         & m.isKey = k ) )
```

For problem (1), if $e.ownedAttribute$ is empty, the antecedent is not satisfiable for any p, and so no $c : CStruct$ will be created for e.

For problem (2), if $CStruct$ does not have a key attribute, then a different $c : CStruct$ could be created for each different $p : e.ownedAttribute$. Check-before-enforce semantics does not avoid this problem because there may not be a $CMember$ with all the required properties for p, and hence, the c domain will need to be executed for p. However, the least-change semantics for target element resolution for the $c : CStruct$ instantiation would resolve problem (2): new $CStruct$ instances would not be created for different p; instead, the $CStruct$ with $name = e.name$ would be reused and only new $CMember$ instances would be created and added as members of this $CStruct$.

To avoid such semantic problems, specifiers should separate matching on many-valued features into separate rules, in which the owner elements are already bound by a *when* clause. For example, $Class2CStructV2$ should be restructured as two rules, $Class2CStruct$ and:

```
top relation MapProperty2CMember
{ enforce domain design e : Entity
   { ownedAttribute = p : Property
     { name = n1, isUnique = k } };
 enforce domain C c : CStruct
   { members = m : CMember { name = n1,
   isKey = k } };
 when { Class2CStruct(e,c) }
}
```

This avoids the problem in the same manner as $NonTopProperty2CMember$: the e and c variables are both bound prior to the quantifiers on p and m, resulting in the logical form $\forall e \forall c \forall p \exists m$ instead of $\forall e \forall p \exists c \exists m$.

In general, the semantics can be used to identify anomalies which can indicate specification errors. In our tools, we include static checks for the correctness conditions (a) to (c) and (e), including flaws such as the use of calls to top relations in where clauses, unused non-top relations, and updates to the same data in the target domains and the where clause of a relation. Figure 8 shows an example of analysis of condition (b) and Fig. 9 an example of analysis of condition (e). (d) cannot be statically checked, only by runtime monitoring.

Confluence analysis identifies cases of non-determinism where the effect of a transformation depends upon the order in which source elements are matched by a relation. This arises in particular if there is a conflict of kind (c) of Sect. 5.2 between one application of the relation and another. For example, if key attributes aId, bId are used for a source class A and target class B involved in a relation:

```
top relation A2B
{ enforce domain src a : A { aId = v, ... };
 enforce domain trg b : B { bId = w, ... };
}
```

then assignments of a constant to bId, such as $bId = $ “1″, are detected as potential confluence errors. These are also cases of potential semantic conflicts (c) within a single relation (the QVT-R semantics of [30] only considers semantic conflicts between different relations).

Assignments of (values of) non-keys $A::att$ to key attributes $B::bId$ are also a potential confluence error because different A instances $a1$, $a2$ can have the same att values and hence match to the same B instance. If other features of $a1$, $a2$ have different values and are used to update B features, then a conflict of type (c) arises. Least-change semantics is also prone to this kind of error. Potential conflicts of type (b) between relations can also be detected by our analysis: a warning is issued if two different top relations update the same features of the same target class (Fig. 8).

A more subtle confluence problem arises with specifications of the form

```
top relation R
{ enforce domain src a : A
   { br = bx : B {} };
 enforce domain trg a1 : A1
  { b1r = b1x : B1 { b2r = b2x :
      B2 {} } };
 when { A2A1(a,a1) and B2B2(bx,b2x) }
}
```

where br and $b2r$ are *-multiplicity references and $b1r$ is a 1-multiplicity reference. The specification requires that for every bx in $a.br$, there exists a corresponding $b2x$ in $a1.b1r.b2r$. The semantics θ_R of the relation R has the form

```
(a,a1) : A2A1$trace & (bx,b2x) : B2B2
    $trace & bx : a.br =>
      B1->exists
     ( b1x | a1.b1r = b1x & b2x : b1x.b2r )
```

Assuming that there are no key features, there are three different options for target resolution of $b1x$: (i) always create new $B1$ instances $b1x$ for each different $bx : a.br$; (ii) check-before-enforce; (iii) least-change check-before-enforce. Option (i) is incorrect, because the assignment $a1.b1r = b1x$ will overwrite any previous assignment to $a1.b1r$ and invalidate previously established $b2x' : a1.b1r.b2r$ for $bx' \neq bx$ and $(bx', b2x') : B2B2\$trace$. Check-before-enforce semantics only avoids the creation of $b1x$ if $b2x$ is already in $a1.b1r.b2r$. Hence overwriting can also occur in this situation. However, the least-change semantics (iii) does give correct behaviour. Using $B1 \rightarrow existsLC(b1x$, an instance $b1x : B1$ with $a1.b1r = b1x$ is only created once, then subsequently this instance is looked-up and $b2x : b1x.b2r$ is established for $b1x$. This avoids the above confluence problem. Thus, using our extended QVT-R notation, the relation should be written as:

```
top relation R
{ enforce domain src a : A
   { br = bx : B {} };
 enforce domain trg a1 : A1 { b1r = b1x
  <:= B1 { b2r = b2x : B2 {} } };
 when { A2A1(a,a1) and B2B2(bx,b2x) }
}
```

Syntactic confluence checks for the default $exists$ semantics are defined in [24] and are implemented in the tools.

7 Design patterns

Several model transformation design patterns from [20,25] are particularly useful in structuring QVT-R transformations in order to reduce semantic flaws and increase capabilities for bidirectional execution:

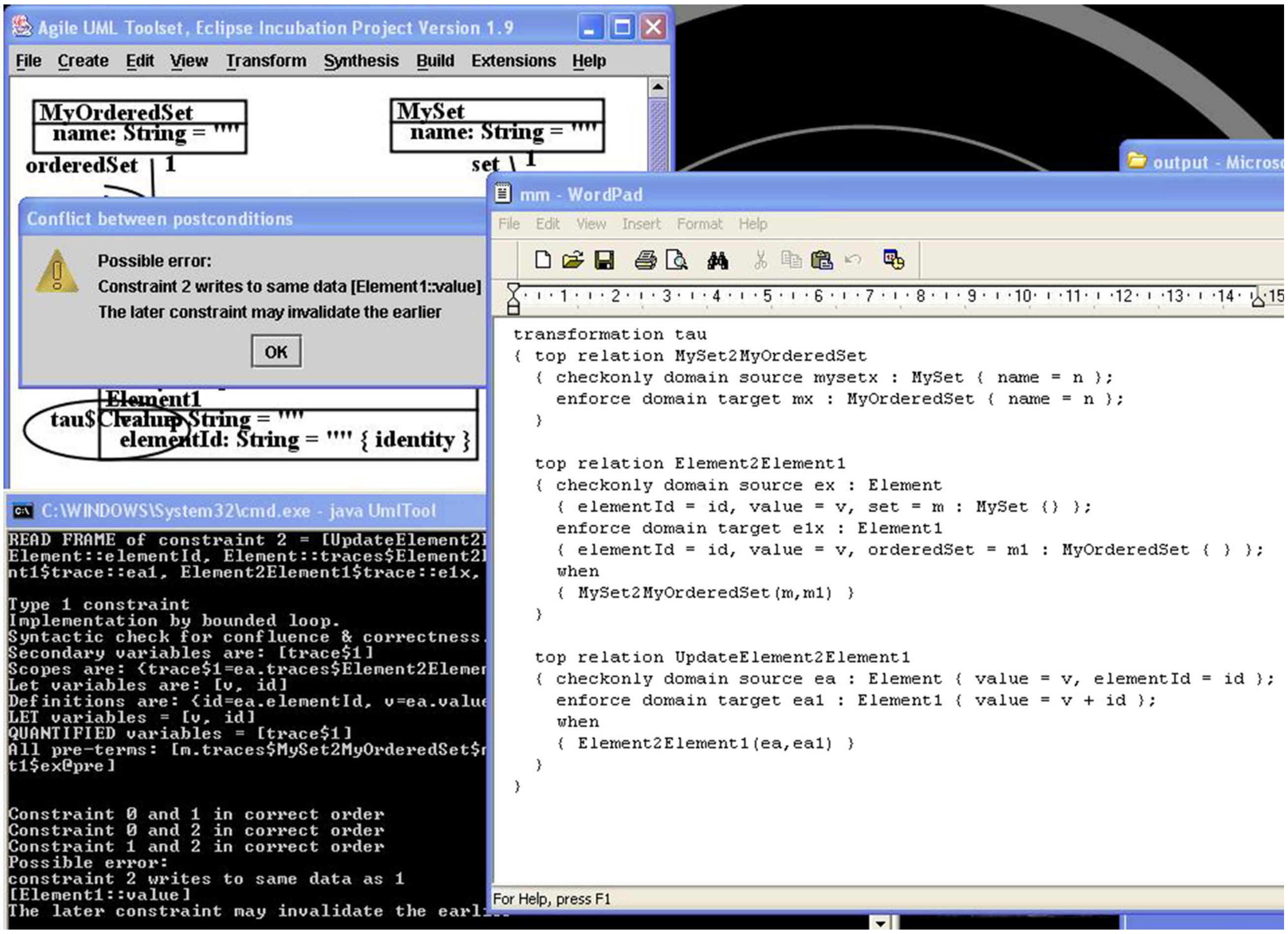
- *Map objects before links*: used to avoid mutual/cyclic dependencies between relations and to separate out the mapping of collection-valued references. The pattern relies on key-based or mandatory-create target resolution, in order to impose a 1–1 mapping of elements in the ‘map’ phase (e.g. $Model2Program$ in Sect. 2).
- *Lens*: used to provide incremental bidirectional execution in cases where the forward map ignores existing target data. This is used in cases where a target data feature g of instances $t : T$ of entity type T can be computed in the forward direction as a function of source features $f_1, \ldots, f_n$ of instances $s_1, \ldots, s_n$ of source entity types $S_1, \ldots, S_n$:

  $$t.g = get(s_1.f_1, ..., s_n.f_n)$$

  In the reverse direction, put_i functions update the f_i based on the initial values f_i@pre of these features and on the target data g:

  $$s_i.f_i = put_i(t.g, s_1.f_1@pre, ..., s_n.f_n@pre)$$

  In QVT-R, the equations can be placed in a relation $where$ clause, the assignment to g will be effective for update only in the source-target execution direction, whilst the assignments to the f_i will be effective only in the target-source execution direction. The @pre suffix is used for the f_i so that correctness condition (a) is not violated in this direction. To avoid conflicting updates in the put assignments, each $s : S_i$ for any i should be related to only one $t : T$.


Agile UML Toolset, Eclipse Incubation Project Version 1.9
File Create Edit View Transform Synthesis Build Extensions Help
MyOrderedSet
name: String = ""
orderedSet 1
MySet
name: String = ""
set 1
Element1
tau$Cleanup String = ""
elementId: String = "" { identity }
mm - WordPad
File Edit View Insert Format Help
output - Micros
For Help, press F1
C:\WINDOWS\System32\cmd.exe - java UmlTool

Conflict between postconditions

Possible error:
Constraint 2 writes to same data [Element1::value]
The later constraint may invalidate the earlier

OK

```
transformation tau
{ top relation MySet2MyOrderedSet
  { checkonly domain source mysetx : MySet { name = n };
    enforce domain target mx : MyOrderedSet { name = n };
  }

  top relation Element2Element1
  { checkonly domain source ex : Element
    { elementId = id, value = v, set = m : MySet {} };
    enforce domain target e1x : Element1
    { elementId = id, value = v, orderedSet = m1 : MyOrderedSet { } };
    when
    { MySet2MyOrderedSet(m,m1) }
  }

  top relation UpdateElement2Element1
  { checkonly domain source ea : Element { value = v, elementId = id };
    enforce domain target ea1 : Element1 { value = v + id };
    when
    { Element2Element1(ea,ea1) }
  }
}
```

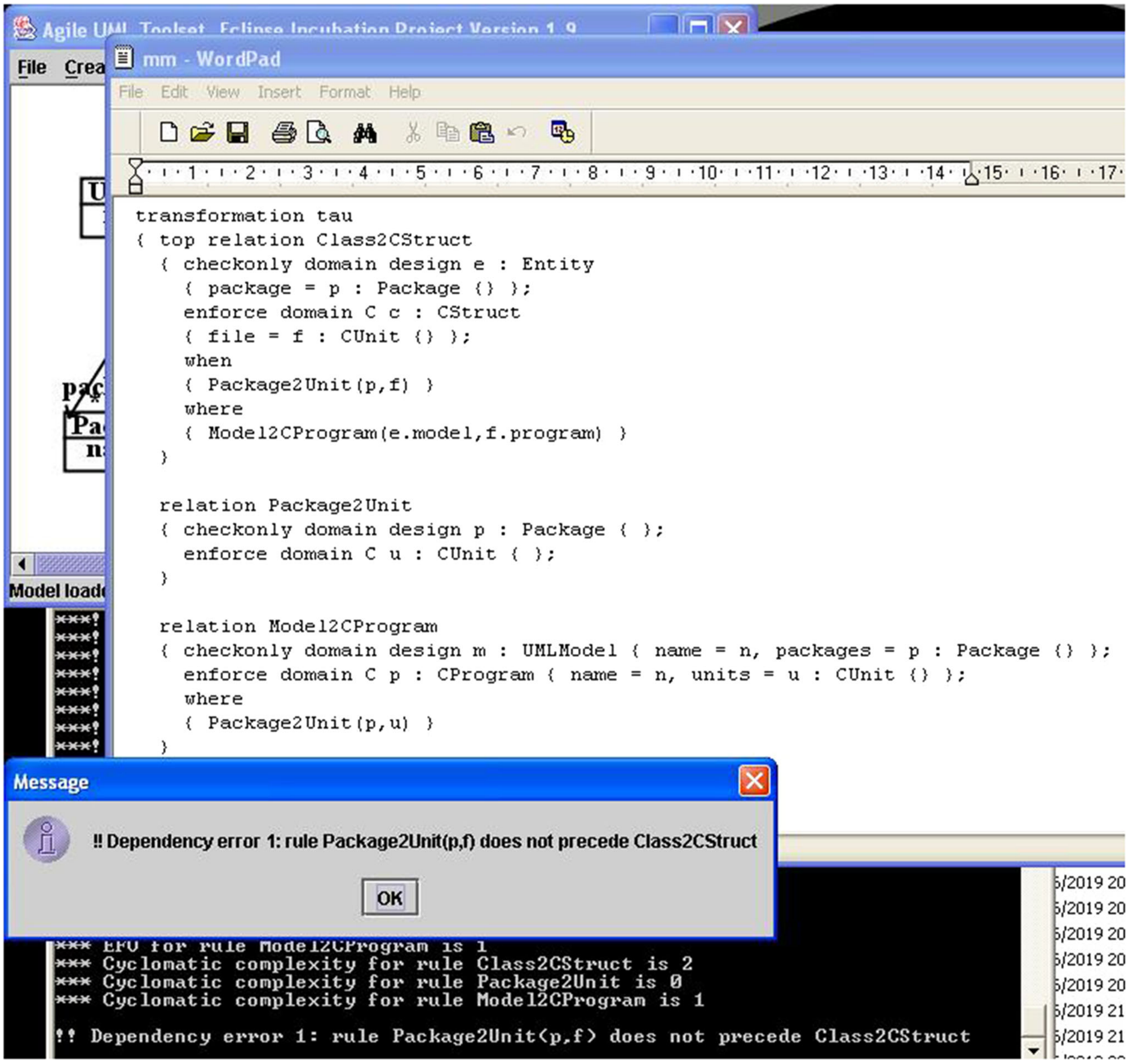
```
READ FRAME of constraint 2 = [UpdateElement2I
Element::elementId, Element::traces$Element2I
nt1$trace::ea1, Element2Element1$trace::e1x,

Type 1 constraint
Implementation by bounded loop.
Syntactic check for confluence & correctness.
Secondary variables are: [trace$1]
Scopes are: {trace$1=ea.traces$Element2Elemen
Let variables are: [v, id]
Definitions are: {id=ea.elementId, v=ea.value
LET variables = [v, id]
QUANTIFIED variables = [trace$1]
All pre-terms: [m.traces$MySet2MyOrderedSet$r
t1$ex@pre]


Constraint 0 and 1 in correct order
Constraint 0 and 2 in correct order
Constraint 1 and 2 in correct order
Possible error:
constraint 2 writes to same data as 1
[Element1::value]
The later constraint may invalidate the earl...
```

Fig. 8 Semantic analysis of QVT-R: update conflicts

- *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: $A \longrightarrow_r B$ in SL is elaborated to $A1 \longrightarrow_{r1} C \longrightarrow_{r2} B1$ in TL, so that r is represented by the composition $r1.r2$. 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 τ 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



```
transformation tau
{ top relation Class2CStruct
  { checkonly domain design e : Entity
    { package = p : Package {} };
    enforce domain C c : CStruct
    { file = f : CUnit {} };
    when
    { Package2Unit(p,f) }
    where
    { Model2CProgram(e.model,f.program) }
  }

  relation Package2Unit
  { checkonly domain design p : Package { };
    enforce domain C u : CUnit { };
  }

  relation Model2CProgram
  { checkonly domain design m : UMLModel { name = n, packages = p : Package {} };
    enforce domain C p : CProgram { name = n, units = u : CUnit {} };
    where
    { Package2Unit(p,u) }
  }
```

!! Dependency error 1: rule Package2Unit(p,f) does not precede Class2CStruct

```
*** EFO for rule Model2CProgram is 1
*** Cyclomatic complexity for rule Class2CStruct is 2
*** Cyclomatic complexity for rule Package2Unit is 0
*** Cyclomatic complexity for rule Model2CProgram is 1
!! Dependency error 1: rule Package2Unit(p,f) does not precede Class2CStruct
```

Agile UML Toolset Eclipse Incubation Project Version 1.9
mm - WordPad
File Edit View Insert Format Help
Message
OK
Model load

Fig. 9 Semantic analysis of QVT-R: relation dependencies

variables $s1 : S$, $s2 : S$—so that each pair $x1$, $x2$ of distinct S elements is only bound in one way to the si, 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 ∀ 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 EFI 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 traces. 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-

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)	

tion (in UML and OCL) from the standard is lower than for the other approaches.

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 (τ) for the fully bx relations, one (σ) for the specific forward relations and another (ρ) for the specific reverse relations. Transformation extension could be used to combine τ with σ and τ with ρ.

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
UML to Python	Bidirectional	Incremental	Yes	Recursive *-accumulation Auxiliary metamodel Map objects before links
Hsm2nhsm	Partly separate forward/reverse	Incremental	Yes	Recursive *-accumulation Entity merging/ splitting (horizontal) Map objects before links
Person migration	Bidirectional	Incremental	Yes	Lens
Weighted/ unweighted Petri nets	Bidirectional	Incremental	Yes	Introduce/remove intermediate classes Map objects before links
Unordered/ ordered sets	Bidirectional	Incremental	Yes	Map objects before links Object indexing
Migration of bags	Bidirectional	Batch	Yes	Auxiliary models Object indexing Lens; Introduce/ remove intermediate class; Map objects before links
Expression trees/ dags	Mutually inverse forward/reverse	Batch	No	Deletion by selective copy
Gantt2CPM	Bidirectional	Incremental	Yes	Entity merging splitting (horizontal) Introduce/remove intermediate class 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].

Table 9 Semantic translation approaches for QVT-R semantic analysis

Aspect	CPN [8]	Mu calculus [2]	UML-RSDS
Modes supported	batch, merging checkonly, enforce	batch, incremental checkonly, enforce	batch, incremental, update-in-place, checkonly, enforce
Target resolution	Key-based	Key-based, check-before-enforce	Key-based, least-change, mandatory create, check-before-enforce
Traces	Non-persistent	None	Persistent/non-persistent
Execution	Simulation via CPN tools	Simulation via model-checker	Code-generation in 3GL
Analysis	Termination, confluence, consistency, debugging	Debugging	Confluence, consistency, debugging
Precise correctness criteria	√	×	√
Restrictions	No calling cycles, OCL restrictions	No element creation in non-top relations	No calling cycles

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.

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 τ, 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.

$\theta_R(m, vars)$ is defined as for the separate-models case, but $Pres_R(m)$ is redefined as:

$$
\begin{array}{l}
R\$trace@pre ::\\
\quad if\ guard_R(m)@pre\ then\ \theta_R(m)'\\
\quad else\ self{\rightarrow}isDeleted()\\
\quad endif
\end{array}
$$

where $\theta_R(m)'$ has antecedent $(cpreds(sdom, ovars_R)\ \&\ whenp)@pre$. We use this prestate version of the antecedent of θ_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:

$$
\begin{array}{l}
::\\
\quad (cpreds(sdom, \{\})\ \&\ whenp)@pre\ \&\\
\quad not(R\$trace@pre{\rightarrow}exists(tr|\&_{sv \in svars_R} tr.sv\\
= sv))\ \implies\\
\qquad\qquad\qquad epreds(tdom, whenvars_R \cup\\
sourcevars_R)\ \&\ wherepx
\end{array}
$$

The $Cleanup_E$ constraints are defined as in the separate-models case, except that the Ri range over all relations with some $e : E$ variable in their $tvars_{Ri}$ set (instead of $outvars_{Ri}$). 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 $Cleanup_E$ constraints can delete source elements (since the source and target models are the same) and delete traces.

The phases $Pres_\tau(m)$, $Con_\tau(m)$, $Cleanup_\tau(m)$ are then defined from the $Pres$, Con and $Cleanup$ 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:

```
top relation R
{ checkonly domain design e : Entity {};
  enforce domain design e : Entity {}
  { e.stereotypes->includes("COM+") };
}
```

For this rule, $Pres_R(design)$ and $Con_R(design)$ are:

```
R$trace@pre::
  if e : Entity@pre
  then
    e.stereotypes->includes("COM+")
  else
    self->isDeleted()
  endif

::
  e : Entity@pre & not(R$trace@pre->
      exists(tr | tr.e = e))   =>
                    e.stereotypes->
     includes("COM+") &
                    R$trace->exists
    ( tr | tr.e = e )
```

This transformation does not need to be iterated, because after one iteration it ensures that $\theta_R(design)$ is true for all model instances. Notice that attributes such as *name* and *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:

```
top relation R
{ checkonly domain graph n : Node {};
  enforce domain graph n : Node {};
  when { n.neighbours->size() = 1 }
}
```

where graph nodes have a set $neighbours : Set(Node)$ of neighbours, a $name : String$ and a unique $id : String$. Such transformations may require repeated execution. In this example, $Con_R(graph)$ will not be enabled on n which have $n.neighbours@pre{\rightarrow}size() \neq 1$; hence, such nodes will not be recorded in $R\$trace$ and will be deleted by $Cleanup_{Node}(graph)$. 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.

$Con_R(graph)$ is:

```
::
  n : Node@pre & n.neighbours@pre->
```

```
   size() = 1 &
  not(R$trace@pre->exists
    ( tr | tr.n = n)) =>
       R$trace->exists( tr | tr.n = n )
```

This places any node with 1 neighbour into the trace.

$Pres_R(graph)$ is:

```
R$trace@pre::
  if n : Node@pre & n.neighbours@pre->
  size() = 1
  then
    n.neighbours@pre->size() = 1 => true
  else self->isDeleted()
  endif
```

A node n may be in the trace because on the previous iteration it had one neighbour in the $Pres_\tau$ phase, but after $Cleanup_\tau$ it does not. $Pres_R(graph)$ will consequently delete the trace containing n, and n will be removed by the following $Cleanup_{Node}(graph)$. The same actions are taken if $n.neighbours$ has been changed by an incremental modification of the model to falsify $n.neighbours.size = 1$.

The consistency properties (1), (2) and (3) of Sect. 5.4 also apply to update-in-place transformations. (1) is established by the $Cleanup_\tau(m)$ constraints, and (3) is redundant because the model has been restricted to elements which are in traces. (2) need not be established by a single iteration. For example, in the $Node$ case, some remaining node may have $neighbours.size \neq 1$ due to deletions carried out by $Cleanup_\tau(m)$. Thus, the phases $Pres_\tau(m)$; $Con_\tau(m)$; $Cleanup_\tau(m)$ need to be iterated until both (1) and (2) hold. In such cases, the annotation @iterated should be attached to the transformation.

The iteration of the transformation is defined by the activity of τ':

```
$while\ not(2)$
$do$
    $(Pres_\tau(m); Con_\tau(m); Cleanup_\tau(m))$
```

Proof of termination will generally require the use of some variant expression $v : \mathbb{N}$, whose value is strictly reduced by each iteration, as in [21]. In the above example, the size of $Node.allInstances()$ could be used as a variant. Another possibility would be to allow the number of iterations to be manually controlled. In this case, the condition (2) could be tested to determine if the execution is complete.

Examples of update-in-place execution are given in the $qvt2umlrsds$ dataset [27], including an update-in-place solution to the tree to dag case of [36].

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 $cpred/epred$ on individual relation domains.

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

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

If $expr$ is an $ObjectTemplateExp$ ote of the form

```
e : E { f1 = val1, ..., fn = valn } { P }
```

then the first result p of $cpred(e, ote, bound)$ is formed as a conjunction of the predicates of $cpred(e, fi = vali, bound_i)$ for each $PropertyTemplateItem$ $fi = vali$, where $bound_1 = bound \cup \{e\}$:

- If $vali$ is itself an $ObjectTemplateExp$, such as $x : F\ \{g1 = w1, \ldots, gm = wm\}$, then the predicate of $cpred(e, fi = vali, bound_i)$ is $x = e.fi$ & $x : F$ conjoined with the predicate result q of $cpred(x, vali, bound_i \cup \{x\})$ if fi is single-valued (its multiplicity is 1), or $x : e.fi$ & $x : F$ & q if fi is collection-valued (multiplicity $\neq 1$). The type declaration $x : F$ is omitted if $x \in bound_i$, 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 $bound_{i+1}$ of $cpred(e, fi = vali, bound_i)$ includes x and $bound_i$.
- Otherwise, if $vali$ is an unbound variable, $vali \notin bound_i$, and fi is single-valued, $cpred(e, fi = vali, bound_i)$ is $(vali = e.fi, bound_i \cup \{vali\})$. If fi is collection-valued, the predicate is $vali : e.fi$. $bound_{i+1} = bound_i \cup \{vali\}$.
- If $vali$ is a bound variable or other expression, the result is $(e.fi = vali, bound_i)$ for single-valued fi or $(vali : e.fi, bound_i)$ for collection-valued fi. That is, $bound_{i+1} = bound_i$

In the case of a set-typed collection template expression cte:

```
s : Set(E) { x ++ rest }{P}
```

$cpred(s, cte, bound)$ is

$(s : Set(E)\ \&\ x : s\ \&\ rest = s - Set\{x\}\ \&\ P,$
$bound \cup \{s, x, rest\})$

Similarly for a sequence-typed template ste:

```
s : Sequence(E) { x ++ rest }{P}
```

$cpred(s, ste, bound)$ is

$(s : Sequence(E)\ \&\ s{\rightarrow}size() > 0\ \&\ x = s{\rightarrow}at(1)$
$\&\ rest = s{\rightarrow}tail()\ \&\ P, bound \cup \{s, x, rest\})$

For the domain root variable e, the predicate $e : E$ is added as the first conjunct of the predicate result of $cpred(e, e : E\{...\}\{P\}, bnd)$. Any additional predicate P of the domain is added as the final conjunct of the predicate.

For target enforce domains, $epreds([], bounds)$ is $true$, whilst $epreds([d]^\frown doms, bounds)$ is $p\ \&\ epreds(doms, b)$ where $(p, b) = epred(d.rootVariable, d.pattern.templateExpression, bounds \cup \{d.rootVariable\})$.

Primitive domains cannot be targets, and collection templates cannot be used in target domains, so d is a relational domain with an object template expression ote. ote is interpreted as a constraint succedent predicate as the predicate result of $epred(e, ote, bound)$ formed as a conjunction of the predicates of $epred(e, fi = vali, bound_i)$ for the $PropertyTemplateItem$ elements $fi = vali$ of ote, where $bound_1 = bound \cup \{e\}$:

- If $vali$ is itself an $ObjectTemplateExp$, such as $x : F\{g1 = w1, \ldots, gm = wm\}$, then $epred(e, fi = vali, bound_i)$ is $(F{\rightarrow}exists(x|e.fi = x\ \&\ p), bound_{i+1})$ where $(p, bound_{i+1}) = epred(x, vali, bound_i \cup \{x\})$ if fi is single-valued, or $(F{\rightarrow}exists(x|x : e.fi\ \&\ p), bound_{i+1})$ if fi is collection-valued. The quantification on x is omitted if x is already in $bound_i$. The $existsLC$ quantifier is used instead of $exists$ if the instantiation operator is $<:=$.
- Otherwise, if fi is single-valued, $epred(e, fi = vali, bound_i)$ is $(e.fi = vali, bound_i)$. If fi is collection-valued, $epred(e, fi = vali, bound_i)$ is $(vali : e.fi, bound_i)$.

The root variable e of the target domain is $exists$-quantified or $existsLC$-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 $epred(e, e : E\{...\}\{P\}, bound)$. The $exists$ or $existsLC$ quantifiers introduced in $epred$ have their scope extended over the remainder of the logical interpretation of $Pres_R$, including the $where$ predicate interpretation[4].

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

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. $var(P)$ is the set of all features and entity type names used in P, likewise for $var^*(P)$. $V{\downarrow}v$ is V with pairs (x, f) removed, where $x \in v$.

In computing $wr(P)$, 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 $E{\rightarrow}exists(x|Q)$ and $E{\rightarrow}existsLC(x|Q)$ in Table 10.

Deleting an instance x of entity type E by $x{\rightarrow}isDeleted()$ 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. $del(x)$ and $del^*(x)$ are the corresponding sets of features and class names potentially written by deletion propagation from x.

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

C.2 Design synthesis

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

Table 11 shows the main cases of the $stat$ 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 $E{\rightarrow}exists(x|P)$ and $F{\rightarrow}exists(y|Q)$ to $E{\rightarrow}exists(x|P\ \&\ F{\rightarrow}exists(y|Q))$ recursively, and similarly for ${\rightarrow}existsLC$.

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 $=$, $:$, $/:$, $<:$, $E[]$, $\rightarrow includes$, $\rightarrow includesAll$, $\rightarrow excludesAll$, $\rightarrow excludes$, $\rightarrow isDeleted$	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: $var^*(P)$	{}
$e1 : e2.r$ $e2.r\rightarrow includes(e1)$ r collection-valued $e1$, $e2$ single-valued	$var(e1) \cup var(e2)$	$\{r\}$	$var^*(e1) \cup var^*(e2)$	$\{(e2, r)\}$
$e2.r\rightarrow excludes(e1)$ $e1\ /:\ e2.r$ r collection-valued $e1$, $e2$ single-valued	$var(e1) \cup var(e2)$	$\{r\}$	$var^*(e1) \cup var^*(e2)$	$\{(e2, r)\}$
$e1.f = e2$ $e1$ single-valued	$var(e1) \cup var(e2)$	$\{f\}$	$var^*(e1) \cup var^*(e2)$	$\{(e1, f)\}$
$e2.r\rightarrow includesAll(e1)$ $e1\ <:\ e2.r$ r, $e1$ collection-valued $e2$ single-valued	$var(e1) \cup var(e2)$	$\{r\}$	$var^*(e1) \cup var^*(e2)$	$\{(e2, r)\}$
$e2.r\rightarrow excludesAll(e1)$ r, $e1$ collection-valued $e2$ single-valued	$var(e1) \cup var(e2)$	$\{r\}$	$var^*(e1) \cup var^*(e2)$	$\{(e2, r)\}$
$E[e1]$	$var(e1) \cup \{E\}$	{}	$var^*(e1) \cup \{E\}$	{}
$E\rightarrow exists(x \vert Q)$ (E concrete class)	$rd(Q)$	$wr(Q) \cup \{E\} \cup$ all superclasses of E	$rd^*(Q)$	$wr^*(Q) \cup \{E\} \cup$ all superclasses of E
$E\rightarrow existsLC(x \vert Q)$ (E concrete class)	$rd(Q)$	$wr(Q) \cup \{E\} \cup$ all superclasses of E	$rd^*(Q)$	$wr^*(Q) \cup \{E\} \cup$ all superclasses of E
$E\rightarrow forAll(x \vert Q)$	$rd(Q) \cup \{E\}$	$wr(Q)$	$rd^*(Q) \cup \{E\}$	$wr^*(Q)$
$x\rightarrow isDeleted()$ x single-valued, of class type E	$var(x)$	$\{E\} \cup del(x)$	$var^*(x)$	$\{E\} \cup del^*(x)$
$C \implies Q$	$var(C) \cup rd(Q)$	$wr(Q)$	$var^*(C) \cup rd^*(Q)$	$wr^*(Q)$
Q & R	$rd(Q) \cup rd(R)$	$wr(Q) \cup wr(R)$	$rd^*(Q) \cup rd^*(R)$	$wr^*(Q) \cup wr^*(R)$
$Q\ or\ R$	$rd(Q) \cup rd(R)$	$wr(Q) \cup wr(R)$	$rd^*(Q) \cup rd^*(R)$	$wr^*(Q) \cup wr^*(R)$
$Q\ xor\ R$	$rd(Q) \cup rd(R)$	$wr(Q) \cup wr(R)$	$rd^*(Q) \cup rd^*(R)$	$wr^*(Q) \cup wr^*(R)$
$if\ E\ then\ E1$ $else\ E2\ endif$	$var(E) \cup$ $rd(E1) \cup rd(E2)$	$wr(E1) \cup wr(E2)$	$var^*(E) \cup$ $rd^*(E1) \cup rd^*(E2)$	$wr^*(E1) \cup$ $wr^*(E2)$
$let\ v\ in\ P\ be\ Q$	$(var(P) - v) \cup$ $(rd(Q) - v)$	$wr(Q)$	$(var^*(P){\downarrow}v) \cup$ $rd^*(Q){\downarrow}v$	$wr^*(Q)$

domain object variable v. In UML-RSDS, construction of objects of concrete class E possessing a key is performed by the $createByPKE(keyvalue)$ operation, whilst creation of objects of concrete classes E without keys is performed by $createE()$. $createByPKE(v)$ only creates a new E instance if there is not already an instance $E[v]$ of E with key value v.

Updates to association ends may require additional further updates to opposite association ends, updates to class extents or to features may require further updates to derived and other data-dependent features, and so forth. These updates are all included in the $stat$ activity. In particular, for $x \rightarrow isDeleted()$, x is removed from every association end and entity type extent in which it resides, and further cascaded deletions may occur if the association ends are mandatory/composition ends. The or and xor constructs are typically used in cases such as $P\ or\ Q$ where Q is an alternative to be established if P fails to be established by $stat(P)$ or $stat_{LC}(P)$. For $P\ xor\ Q$, a normalisation should exist for $not(Q)$.

The clauses for $X \rightarrow exists(x | x.id = v\ \&\ P1)$ and $X \rightarrow existsLC(x | x.id = v\ \&\ P1)$ with id an identity attribute of X test for existence of an x with $x.id = v$ before creating such an object: this has implications for efficiency but is necessary for correctness: two distinct X elements with the same key value should not exist.

$stat_{LC}(P)$ gives a "least-change" procedural interpretation of expressions P: an update is only performed by this interpretation to establish P if P does not already hold, or if the update would make no change to data in the case P holds.

$stat_{LC}(v = e)$ is the same as $stat(v = e)$, $v{:=}e$.

$stat_{LC}(v : e)$ is the same as $stat(v : e)$ for sets e, but for sequences it is

$$if\ e \rightarrow includes(v)\ then\ skip\ else\ stat(v : e)$$

Similarly, for <:, / : and / <:. In the special case of 0..1 multiplicity features r, $stat_{LC}(y : x.r)$ is

$$if\ x.r \rightarrow isEmpty()\ then\ x.r\ := \ x.r \rightarrow including(y)\ else\ skip$$

$stat_{LC}(P\ \&\ Q)$ is $stat_{LC}(P); stat_{LC}(Q)$ under the assumption that $P \implies [stat_{LC}(Q)]P$. This is ensured if $wr(Q)$ is disjoint from $wr(P)$ and $rd(P)$.

$stat_{LC}(if\ C\ then\ P\ else\ Q\ endif)$ is

```
if C
then statLC(P)
else statLC(Q)
```

under the assumptions $C \implies [stat_{LC}(P)]C$ and $\neg C \implies [stat_{LC}(Q)]\neg C$. These are ensured if $wr(P)$ and $wr(Q)$ are disjoint from $var(C)$.

$stat_{LC}(P\ or\ Q)$ is

```
statLC(P) ;
if not(P) then statLC(Q) else skip
```

$stat_{LC}(P\ xor\ Q)$ is

```
if P & Q
then statLC(not(Q))
else
  if not(P) & not(Q)
  then statLC(P) else skip
```

For $s \rightarrow forAll(x|P)$, $stat_{LC}$ is defined as $for\ x\ :\ s\ do\ stat_{LC}(P)$.

For existential quantifiers $E \rightarrow existsLC(e|P_1\ \&\ \ldots\ \&\ P_n)$, their $stat$ or $stat_{LC}$ effect only creates a new e in cases where there is no existing $e : E$ that satisfies P partially or completely. In the case of partial satisfaction, the updates only for the unsatisfied conjuncts are carried out.

If E has an identity attribute pk and a conjunct P_i is of the form $e.pk = value$, then $stat(E \rightarrow existsLC(e|P_1\ \&\ \ldots\ \&\ P_n))$ is

```
var e : E;
e := createByPKE(value);
statLC(Pred)
```

Where $Pred$ is $P_1\ \&\ \ldots\ \&\ P_n$ with P_i omitted.

Otherwise, $stat(E \rightarrow existsLC(e|P_1\ \&\ \ldots\ \&\ P_n))$ has the form:

```
var e : E;
var eset : Set(E);
eset := E.allInstances();
if eset->isEmpty()
then
  e := createE();
  statLC(P1 & ... & Pn)
else
  (e := eset->any(true);
   eset := eset->select(P1);
   if eset->isEmpty()
   then
     e := createE();
     statLC(P1 & ... & Pn)
   else
     (e := eset->any(true);
      eset := eset->select(P2);
      if eset->isEmpty()
      then
        statLC(P2 & ... & Pn)
      else
        ... case for 3 ... ) )
```

The general case for $k \geq 2$, $k < n$ is

```
e := eset->any(true);
eset := eset->select(Pk);
```

Table 11 Definition of $stat(P)$

P	$stat(P)$	Condition
$true$	skip	
$x = e$	$x{:=}e$	x is assignable, $x \notin var(e)$
$e : x$ $x{\rightarrow}includes(e)$	$x{:=}x{\rightarrow}including(e)$	x is assignable, collection-valued, $x \notin var(e)$
$e\ /:\ x$ $x{\rightarrow}excludes(e)$	$x{:=}x{\rightarrow}excluding(e)$	x is assignable, collection-valued, $x \notin var(e)$
$e <: x$ $x{\rightarrow}includesAll(e)$	$x{:=}x{\rightarrow}union(e)$	x is assignable, collection-valued, $x \notin var(e)$
$e\ /<:\ x$ $x{\rightarrow}excludesAll(e)$	$x{:=}x - e$	x is assignable, collection-valued, $x \notin var(e)$
$x{\rightarrow}isDeleted()$ (single object x)	;-composition of $E{:=}E{\rightarrow}excluding(x)$ and $y.r{:=}y.r{\rightarrow}excluding(x)$	Each class E containing x each association end $y.r$ containing x
$obj.op(e)$	$obj.op(e)$	Single object obj
$objs.op(e)$	for $x : objs$ do $x.op(e)$	Collection $objs$
$P1\ \&\ P2$	$stat(P1); stat(P2)$	$wr(P2) \cap wr(P1) = \{\}$ $wr(P2) \cap rd(P1) = \{\}$

$P1\ or\ P2$	$stat(P1)$;
	 if $not(P1)$ then $stat(P2)$
	 else skip	

$P1\ xor\ P2$	if $P1\ \&\ P2$ then $stat(not(P2))$ else
	 if $not(P1)\ \&\ not(P2)$
	 then $stat(P1)$ else skip
		$not(P2)$ can be normalised

$E{\rightarrow}exists(x \mid x.id = v$
 $\&\ P1)$
	if $E.id{\rightarrow}includes(v)$
	 then var $x : E := E[v]; stat(P1)$
	 else (var $x : E := createByPKE(v)$;
	 $stat(P1))$
		E is a concrete class with $E{\rightarrow}isUnique(id)$

$E{\rightarrow}exists(x \mid P1)$	(var $x : E := createE()$; $stat(P1))$	E is a concrete class, $P1$ not of form $x.id = v\ \&\ P2$ for unique id attribute of E

$e{\rightarrow}exists(x \mid x.id = v$
 $\&\ P1)$
	if $e{\rightarrow}includes(E[v])$ then
	 (var $x : E := E[v]; stat(P1))$
	 else skip
		Non-writable expression e with element type class E, $E{\rightarrow}isUnique(id)$

$e{\rightarrow}exists(x \mid P1)$	if $e{\rightarrow}notEmpty()$ then (var $x : E := e{\rightarrow}any(true)$; $stat(P1))$ else skip	Non-writable expression e, $P1$ not of above form
$E{\rightarrow}forAll(x \mid P1)$	for $x : E$ do $stat(P1)$	$wr(P) \cap rd(P) = \{\}$
$P1 \implies P2$	if $P1$ then $stat(P2)$ else skip	$P1$ side-effect free $wr(P2) \cap var(P1) = \{\}$

$if\ E\ then\ P1$
 $else\ P2\ endif$
	if E then $stat(P1)$
	 else $stat(P2)$
		E side-effect free
		 $(wr(P1) \cup wr(P2)) \cap var(E) = \{\}$

$let\ v\ be\ P\ in\ Q$	any v where P then $stat(Q)$	$wr(Q) \cap var(P) = \{\}$

```
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 $e{:=}eset{\rightarrow}any(true)$; $result{:=}e$. Otherwise it is

```
e := eset->any(true);
eset := eset->select(Pn);
if eset->isEmpty()
then
  statLC(Pn)
else
  e := eset->any(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:

$$\begin{array}{l} d_S \in S.domain \implies \\ \quad \exists d_R \in R.domain \cdot \\ \qquad d_R.rootVariable.name = d_S.rootVariable.name \wedge \\ \qquad d_R.isCheckable = d_S.isCheckable \wedge \\ \qquad d_R.isEnforceable = d_S.isEnforceable \wedge \\ \qquad d_R.isPrimitive = d_S.isPrimitive \wedge \\ \qquad d_R.typedModel = d_S.typedModel \end{array}$$

The type $d_R.rootVariable.type$ 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 $when$ clause of P is the conjunction $S.when$ and $R.when$, i.e. the pattern formed by concatenating $S.when.predicate$ and $R.when.predicate$ and removing duplicated conjuncts. The same applies for the $where$ clauses. P is abstract if R is abstract, and concrete if R is concrete.

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

$$v = val1 \ \oplus \ p \ = \ p$$

where $p.referredProperty.name = v$.

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 $w.name = v.name$ 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 $where$ conditions of the two templates are conjoined to form the $where$ condition of the result.

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

$$s : Set(E)\{x \mathrel{++} rest\}\{P\}$$

and cte_R:

$$s : Set(F)\{x \mathrel{++} rest\}\{Q\}$$

is defined as $cte_S \oplus cte_R$:

$$s : Set(F)\{x \mathrel{++} rest\}\{P \ and \ Q\}$$

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 τ and σ should have the same set of typed models:

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

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

$$\rho.helpers = \tau.helpers{\rightarrow}union(\sigma.helpers)$$

The relations of ρ are the union of those of τ and σ, but with common-named rules of the two transformations combined using $\oplus$.

$$\begin{aligned}&\rho.rule \ = \ \sigma.rule{\rightarrow}reject(r|\tau.rule.name{\rightarrow}includes\\&(r.name)){\rightarrow}union(\\&\qquad \tau.rule{\rightarrow}reject(r|\sigma.rule.name{\rightarrow}includes\\&(r.name)){\rightarrow}union(\\&\qquad \sigma.rule{\rightarrow}select(r|\tau.rule.name{\rightarrow}includes\\&(r.name)){\rightarrow}collect(r|r \oplus \tau.rule{\rightarrow}any\\&(name = r.name))))\end{aligned}$$

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.wikidot.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. Greenyer, 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 (2014)
9. Hermann, 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). https://github.com/eclipse/agileuml/blob/master/qvt2umlrsds.pdf
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. MOF2 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

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.