DOI:
10.1109/ACCESS.2018.2889399

Citation for published version (APA):
Lano, K., Kolahdouz Rahimi, S., & Yassipour Tehrani, S. (in press). Declarative Specification of Bidirectional Transformations Using Design Patterns. *IEEE Access*, *7*(1), 5222-5249. Article 8587240. https://doi.org/10.1109/ACCESS.2018.2889399

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 6, 2018, accepted December 12, 2018, date of publication December 24, 2018,
date of current version January 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2889399

Declarative Specification of Bidirectional Transformations Using Design Patterns

KEVIN LANO[1], SHEKOUFEH KOLAHDOUZ-RAHIMI[2], AND SOBHAN YASSIPOUR-TEHRANI[1]
[1]Department of Informatics, King's College London, London WC2B 4BG, U.K.
[2]Department of Software Engineering, University of Isfahan, Isfahan 81746-73441, Iran

Corresponding author: Kevin Lano (kevin.lano@kcl.ac.uk)

ABSTRACT Bidirectional transformations (bx) are a specific form of model transformation (MT) used in model-driven engineering to maintain consistency between two models, which may change independently. Currently bx are defined using a number of specialized transformation languages, which have had limited uptake due to complex semantics and poor efficiency. In contrast, unidirectional transformation languages such as ATL have been widely adopted, but require separate forward and reverse transformations to be written to address model synchronization requirements. In this paper, we provide declarative specification techniques for bx, systematically constructed using MT design patterns. We define two approaches to declarative bx definition: 1) by automatically bidirectionalizing unidirectional transformation specifications and 2) by developing specification guidelines for the QVT-R standard language to make it more effective for bx in practice. The approaches are evaluated using a large-scale code-generator bx from UML to ANSI C and other examples. Their semantic validity is demonstrated by rigorous arguments.

INDEX TERMS Bidirectional transformations, design patterns, model transformations, QVT-R, UML-RSDS.

I. INTRODUCTION

Model-driven engineering (MDE) and Model-based development (MBD) have reached a point of significant application in industry, particularly in the automotive industry and more recently in finance. These development approaches use models (such as UML class diagrams) as key artifacts within their processes. In some cases the models are used as the complete definition of an application, with executable code being generated automatically from the models.

Model transformations (MT) are a key element of MDE and MBD approaches for software, supporting code generation, migration, refinement, reverse-engineering and many other capabilities. *Bidirectional* model transformations (bx) are a specific type of MT which provide a means to maintain a consistency relation R between two models which may both change. This is in contrast to *unidirectional* transformations, which establish R by constructing a target model from a source model, and do not subsequently maintain the consistency of these models.

Bx are significant for a wide range of model based engineering scenarios involving model synchronization:

- Incremental execution of transformations, for example, it is desirable for a code generator to only make the minimal necessary changes to generated code if a source model is changed, rather than regenerating the entire code set again.
- Round-trip engineering [28], where either an application specification or design may change and the consistency of the two models at different levels of abstraction must be automatically maintained.
- Automatic consistency maintenance of multiple interrelated models at the same level of abstraction [18], such as UML class diagrams and sequence diagrams, to support agility in model-based development.
- Application and database co-evolution [5].

The motivating case for this paper was a code generator from UML to ANSI C [44]. For realistic applications involving large-scale models with thousands of elements, incremental code generation of C is a practical necessity. Additionally, the use of a bx in this case is motivated by considerations of semantic correctness of the generated code, and by the desirability of round-trip engineering capabilities.

A bx can be used to enforce a relation R between two different representations of similar semantic information. For example, a UML class and a corresponding ANSI C struct. Such a bx is responsible for ensuring that changes to one of these representations is propagated to the other to maintain their mutual consistency. There may however be additional information in one model which has no counterpart in the other. For example, in the well-known Families to Persons bx benchmark [1], the persons model includes *birthday* information, whilst the families model differentiates parents and children. These features are specific to just one of the linked models.

In the theoretical framework of [59], bx are characterized by a binary relation $R : SL \leftrightarrow TL$ between a source language (metamodel) SL and a target language TL. $R(m, n)$ holds for a pair of models m of SL and n of TL when the models consist of data which correspond under R.

The key idea of a bx is that instead of defining two separate transformations from SL to TL and from TL to SL, to establish R, it should be possible to automatically derive from the definition of R both forward and reverse transformations

$$R^{\rightarrow} : SL \times TL \rightarrow TL$$
$$R^{\leftarrow} : SL \times TL \rightarrow SL$$

which establish/maintain R between the two models, given both existing source and target models. This eliminates the risk that forward and reverse transformations could become inconsistent with each other, and reduces the workload involved in maintaining two separate transformation specifications instead of one.

The $R^{\rightarrow}$, $R^{\leftarrow}$ transformations should satisfy two key conditions [46], [59]:

1) *Correctness*: the forward and reverse transformations derived from a relation R do establish R:

$$R(m, R^{\rightarrow}(m, n))$$
$$R(R^{\leftarrow}(m, n), n)$$

for each $m : SL$, $n : TL$.

2) *Hippocraticness*: if source and target models already satisfy R then the forward and reverse transformations do not modify the models [48]:

$$R(m, n) \implies R^{\rightarrow}(m, n) = n$$
$$R(m, n) \implies R^{\leftarrow}(m, n) = m$$

for each $m : SL$, $n : TL$.

Hippocraticness is a global property, in practice a stronger local property is desirable: if any part of a target (source) model is already consistent with the corresponding part of the source (target) model, then neither part should be modified by forward or reverse transformations. We refer to this as *local Hippocraticness*. Further, it is preferable that updates to a model by $R^{\rightarrow}$ or $R^{\leftarrow}$ should be minimal necessary to restore R: this is termed the *principle of least change* [48].

In the following, we will consider only *separate-models* transformations, where source and target models are distinct, and not *update-in-place* transformations, which operate on a single model which is both the source and target.

We describe two alternative approaches to bx definition:

1) In the first approach, unidirectional separate-models transformations τ are defined declaratively using the Object Constraint Language (OCL) [52], and a bx relation R and forward and reverse transformations $R^{\rightarrow}$, $R^{\leftarrow}$ are derived syntactically from the specification of τ.
We consider specifically the unidirectional MT language UML-RSDS [42].
2) In the second approach, a bx τ is defined using a restricted subset of QVT-R [51], with the specification of τ organized so that both $R^{\rightarrow}$ and $R^{\leftarrow}$ exist and satisfy the bx properties wrt R derived from the specification.

An original contribution of our work is an emphasis on the role of design patterns in the definition of τ in order to achieve bx properties.

Section II describes related work, Section III describes UML-RSDS, Section IV describes model transformation and bx design patterns, and Section V defines our approach to bx specification in UML-RSDS. Section VI considers inverse transformations and the bx properties of $R^{\leftarrow}$. Section VII describes a large case study of a bx transformation between UML and C. Section VIII describes our QVT-R based approach to bx definition. Section IX gives an evaluation of the approaches, and Section X gives conclusions. The appendix provides an argument that the bx properties for $R^{\rightarrow}$ are satisfied by transformations defined using our first approach.

II. RELATED WORK

Bx are the subject of extensive theoretical research [10], [12], [46], [58]–[60] and an annual conference, BX. The most well-known transformation language which supports definition of bx is QVT-R [51], which uses an OCL-based expression language. However, QVT-R has had limited uptake because of its complex semantics and specification style, compared to simpler unidirectional MT languages such as ATL [31] and ETL [33]. The semantic basis of the language remains unclear [8], [46], [60]. The combination of bx and update-in-place processing involves aspects such as implicit deletion (if an element is not required to exist by any rule, then it is deleted), which are difficult to use, and are implemented differently by different QVT-R tools [8], [40]. Although the language is intended to be declarative, the meaning of a specification can depend on procedural aspects (rule invocation relationships) in addition to logical predicates. As the examples of [46] show, this leads to specifications which are difficult to understand and liable to subtle errors.

Triple Graph Grammars (TGG) are a graph transformation-based formalism which supports bx definition [14], although with a restricted expression language compared to OCL. Other approaches, such as JTL [11], and extensions of TGG [2], use constraint-based programming techniques to interpret relations $P(s, t)$ between source and target elements as specifications in both forward and reverse directions. Model finding using Alloy is used to bidirectionalize ATL in [46], and to backward propagate view changes in [58]. These approaches have problems of efficiency for large-scale models. An alternative constraint-based approach is to generate constraints restricting target models, instead of the models themselves [13]. This allows greater freedom of choice in the construction of the target model, but requires an additional step to actually derive specific models.

Another research direction has focused on the special case of asymmetric bx known as *lenses*, in which the source to target mapping is independent of the existing target model. The target model $t : TL$ is computed as a view or abstraction $get(s)$ containing partial information from the source model $s : SL$ [6], [15]. A dual map $put : SL \times TL \rightarrow SL$ propagates view changes to the source, such that $put(s, get(s)) = s$ and $get(put(s, t)) = t$. Various semantic and syntactic techniques have been developed for deriving the put function from get in different cases [34], [61].

In previous papers we have briefly introduced some bx design patterns [39] and techniques for bx verification [41]. In this paper we extend these previous works by comprehensively defining an approach to bx specification using UML-RSDS, and we identify additional bx patterns. We also introduce an approach using QVT-R with the bx patterns. We provide rigorous arguments to show the correctness of the UML-RSDS approach, provide an evaluation, and give new examples and extensive details of a large case study.

III. UML-RSDS

In our first approach to bx definition, we look at a unidirectional MT language, UML-RSDS. In this approach, unidirectional transformations τ are specified using UML use cases with OCL postconditions $Post_\tau$ and invariants Inv_τ, then a statically-computed bx relation R_τ for τ is defined as $Post_\tau$ & Inv_τ for τ, together with a forward transformation $\tau^{\rightarrow}$ extending τ, and an inverse $\tau^{\leftarrow}$ of $\tau^{\rightarrow}$. These play the role of $R^{\rightarrow}$ and $R^{\leftarrow}$ in bx theory. The bx relation and transformations are derived from τ using a higher-order transformation. This is generally more efficient than the use of constraint programming and model finding.

UML-RSDS is a precise subset of UML 2 and OCL 2.4, with a formal semantics [35] and an established toolset [36], [42]. We use a variant notation for OCL, with $E.allInstances()$ abbreviated to E for entity types on the lhs of a $\rightarrow$ operator. $x : s$ is used for $s \rightarrow includes(x)$, $s1 <: s2$ for $s2 \rightarrow includesAll(s1)$, & is used for conjunction, $\Longrightarrow$ for implication.[1] Logical expressions P may be used operationally, to stand for behaviors $stat_{LC}(P)$ which establish P (Section III-C). In particular, the predicate $x \rightarrow isDeleted()$ is used to express that the element x is to be removed from all entity types and other collections in which it resides. $s \rightarrow isDeleted()$ for collection s means $s \rightarrow forAll(x | x \rightarrow isDeleted())$. The general *iterate* operator is excluded from our subset, as are tuples and *invalid*. *null* cannot be explicitly referred to, but only tested using $x \rightarrow oclIsUndefined()$.

A. TRANSFORMATION SPECIFICATION

Model transformations τ are specified in UML-RSDS as UML use cases, defined declaratively by two main predicates, expressed in our OCL variant:

1) Postconditions $Post_\tau$ which define the intended effect of the transformation at its termination. These are an ordered conjunction C_1 & ... & C_n of OCL constraints (also termed *rules* in the following) and also serve to define a procedural implementation of the transformation.
2) Invariants Inv_τ which define expected invariant properties which should hold during the transformation execution (their truth is preserved by individual transformation rule applications). The invariants can be derived from $Post_\tau$, or can be specified explicitly by the developer.

From a declarative viewpoint, $Post_\tau$ defines the conditions which should be established by the transformation. From an implementation perspective, the constraints of $Post_\tau$ also define the intended computation steps of the transformation: each computation step is an application of a postcondition constraint (transformation rule) to a specific source model element or to a tuple of source model elements. The computation steps should preserve Inv_τ.

A typical postcondition constraint/rule has the form:

$$S_i ::$$
$$Cond \Longrightarrow Pred$$

where S_i is the *context entity type* of the constraint, *Cond* is the *application condition* of the constraint, and *Pred* its predicate or *succedent*. The logical meaning of the constraint is that for each instance *self* of S_i which exists in the source model, if *self* satisfies *Cond*, then *Pred* is true for *self*. The operational meaning of the constraint is that for each instance *self* of S_i which exists in the source model, if *self* satisfies *Cond*, then *Pred* is made true for *self*, using the activity $stat_{LC}(Pred)$.

Typically, *Pred* will specify the existence of instances t of target entity types T_j via formulae $T_j \rightarrow exists(t | P)$ for concrete entity type T_j, where t's feature values are specified in P.

For example, the simple Families to Persons [1] forward transformation *family2person* (Figure 1 shows the basic metamodels of this transformation) could have two rules:

```
Family::
 m : mother->union(daughters) =>
    Female->exists( p |
        p.name = name + ", " + m.name )

Family::
 m : father->union(sons) =>
    Male->exists( p |
        p.name = name + ", " + m.name )
```

These rules $Post_{family2person}$ express that for each *Family* instance, and for each *FamilyMember* m which is either a father or son of the family, there should exist a corresponding *Male* instance with a name formed as the concatenation of

[1] Using symbols for logical connectives helps to visually distinguish these from the parts of the formulae that they connect.

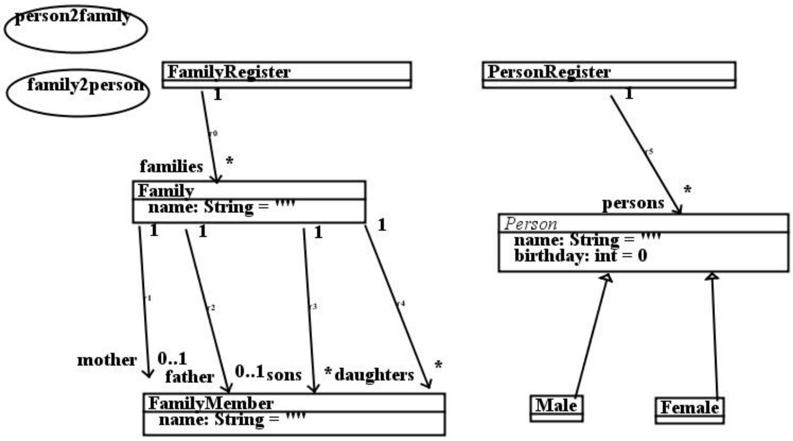


FIGURE 1. Family to Person metamodels.

the family and family member names. Likewise, mothers and daughters are mapped to *Female* instances. The invariants $Inv_{family2person}$ can be derived syntactically from these rules by inverting the quantifiers (Table 5, case 2):

```
Female::
 Family->exists( fx |
   FamilyMember->exists( m |
     m : fx.mother->union(fx.daughters) &
     name = fx.name + ", " + m.name ) )

Male::
 Family->exists( fx |
   FamilyMember->exists( m |
     m : fx.father->union(fx.sons) &
     name = fx.name + ", " + m.name ) )
```

These invariants are preserved by the *family2person* rules: if a new *Female* instance is created by the first rule, then it necessarily satisfies the first invariant because of the context entity type and application conditions of the rule. The new instance does not affect the truth of the second invariant, which only restricts *Male* instances. Likewise, the second rule preserves both invariants.

The importance of transformation invariants in verification is that they can be used to prove transformation *contracts*: conditions P and Q such that, if transformation τ is executed on models satisfying P, it is ensured to terminate in a state with the models satisfying Q [7], [19]. If I is a condition such that P implies I, with I preserved by each rule application of τ, and such that I together with the fact that no rule is enabled implies Q, then the correctness part of the contract is established.

The key idea of our first bx approach is that change-propagation from the source model to the target can be achieved by extending a forward transformation τ by additional constraints/rules derived automatically from Inv_τ. In addition, a reverse transformation $\tau^{\leftarrow}$ can also be derived automatically from Inv_τ. The reverse transformation is based on interpreting the invariants as rules in the inverse direction (notice that the form of the invariants in the *family2person* example is very similar to that of use case postconditions). A rule predicate P such as $t.name = s.name + \text{“*”}$ is converted to an explicit inverse form $P^{\sim}$ such as $s.name = t.name.front$ in order to provide a computationally explicit version of the invariants. This approach therefore achieves the aims of bx in enabling a single transformation specification to be used for both forward and reverse transformations.

B. TRANSFORMATION IMPLEMENTATION

Designs for model transformations and for general applications are defined in UML-RSDS using class diagrams, with UML activities expressing the explicit algorithms of operations and use cases.

The following concrete syntax is used for a subset of UML structured activities:

```
< statement > ::=     < loop > | < creation > |
                      < conditional > |
                      < sequence > | < choice > |
                      < basic >
< loop > ::=          “while” < expression >
                      “do” < statement > |
                      “for” < expression >
                      “do” < statement >
< conditional > ::= “if” < expression >
                      “then” < statement >
                      “else” < basic >
< sequence > ::=      < statement > “;” < statement >
< choice > ::=        < statement > “[]” < statement >
< creation > ::=      “var” <identifier> “:” <identifier>
< basic > ::=         < expression > “:=” < expression > |
                      “skip” |
                      “execute” < expression > |
                      “return” < expression > |
                      “(” < statement > “)” |
                      < call_expression >
```

For use cases, their procedural implementation as an activity is derived as a composition of the activities $stat_{LC}(Cn)$ for each postcondition constraint Cn. $stat_{LC}(Cn)$ is constructed so that it establishes Cn: $[stat_{LC}(Cn)]Cn$, where $[act]P$ is the weakest-precondition of predicate P with respect to statement/activity act. As a predicate transformer, $stat_{LC}(Cn)$ is chosen as a minimal (least-refined) transformer which establishes Cn. In terms of the refinement calculus [50]:

$$[C]P \implies (stat_{LC}(P) \sqsubseteq C)$$

for statements C which do not update more variables than $stat_{LC}(P)$. Intuitively, this means that $stat_{LC}(P)$ performs the minimal necessary data modifications needed to establish P, operating on the write frame $wr(P)$ of P. A definition of $stat_{LC}$ is given in Section III-C. There may not be a unique least-refined statement that establishes a given predicate, in such cases a suitable minimally-refined statement is selected instead.

Data-dependency relations between the *Post* constraints are important in ensuring the correctness of both unidirectional and bidirectional UML-RSDS transformations. Let $rd(Cn)$ denote the read-frame of a constraint Cn, the set of entity type names and data features that it reads, and $wr(Cn)$ denote its write frame. These are defined in [36]. The definition of $wr(P)$ for a predicate P interpreted operationally as $stat_{LC}(P)$, includes features and entity types affected by implicit updates which may arise due to metamodel constraints involving parts of the metamodel explicitly updated by $stat_{LC}(P)$. There are five main cases of such implicit updates:

1) A bi-directional association $A \; {}^{m1}_{ar}\!—\!{}^{m2}_{br} \; B$ between entity types A and B, for any multiplicities $m1$, $m2$: changes to ar will also produce updates to br and vice-versa.
2) An association $A \; {}^{m1}\!—\!{}^{m2}_{br} \; B$: deletion of B instances will remove these from br.
3) An association $A \; {}^{1}\!—\!{}^{m2}_{br} \; B$ with 1-multiplicity end at entity type A: deletion of A instances ax will result in deletion of all B instances linked to ax via br. For example, in Figure 1, deletion of a *Family* instance leads to deletion of its linked *FamilyMember*s.
4) A composition aggregation: deletion of the composite will result in deletion of its linked parts.
5) Creation or deletion of instances of a subclass B of a class A will also result in addition/removal of these instances from the extent of A.

In Figure 1, deletion of *Family* instances will also affect $FamilyRegister :: families$, even though this feature is not explicitly modified by the transformation. Likewise for *Person* and $PersonRegister :: persons$. We say that there is deletion propagation from entity type E_i to entity type E_j if there is a non-empty chain of dependencies of kinds 3, 4 and 5 above between E_i and E_j such that deletion of an E_i instance may cause deletion of an E_j instance.

A dependency ordering $Cn < Cm$ is defined between constraints by $wr(Cn) \cap rd(Cm) \neq \{\}$ "Cm depends on Cn". A use case with postconditions $C_1, \ldots, C_n$ should satisfy the *syntactic non-interference* conditions:

1) If $C_i < C_j$, with $i \neq j$, then $i < j$.
2) If $i \neq j$ then $wr(C_i) \cap wr(C_j) = \{\}$.

Together, these conditions ensure that the activities $stat_{LC}(C_j)$ of subsequent constraints C_j cannot invalidate earlier constraints C_i, for $i < j$.

A constraint is termed a *type 1* constraint if its write and read frames are disjoint. Such constraints usually have an implementation as a bounded loop, as opposed to a fixed-point/unbounded iteration. Both of the rules for Family to Person above are of type 1.

Entity types may have *identity* attributes of String type. These uniquely identify instances of the entity: two different instances must have different values of the attribute. The first such attribute is used as a primary key for the entity type.

If entity type E has a primary key $att : String$, then $E[v]$ denotes the instance x of E with key value $x.att$ equal to v, or the collection of E instances with key value in v, if v is a collection.

C. DEFINITION OF ACTIVITIES FOR PREDICATES

Table 1 gives the definition of $stat_{LC}$ for simple predicates.

TABLE 1. Definition of $stat_{LC}$ for simple predicates.

Predicate P	$stat_{LC}(P)$
$x = e$ Assignable basic expression x	$x := e$
$e : x$ Assignable basic expression x, Set-valued	$x := x \rightarrow including(e)$

$e : x$
Assignable basic expression x,
Sequence-valued
	if $x \rightarrow includes(e)$
	then skip
	else
	$x := x \rightarrow append(e)$

$e \; / : x$ Assignable basic expression x	$x := x \rightarrow excluding(e)$
$e <: f$ Assignable basic expression f	for $x : e$ do $stat_{LC}(x : f)$
$e \; / <: f$ Assignable basic expression f	$f := f - e$

An assignable basic expression x is either a variable, a non-frozen feature f or a basic expression $obj.f$ or $obj.f[i]$ whose modified feature f is non-frozen. $stat_{LC}(x \rightarrow includes(e))$ is the same as $stat_{LC}(e : x)$, and likewise $x \rightarrow excludes(e)$ is considered the same as $e \; / : x$ and $x \rightarrow includesAll(e)$ the same as $e <: x$.

$stat_{LC}(true)$ is *skip*. For $x \rightarrow isDeleted()$, where x is an instance of class E, $stat_{LC}$ removes x from the set of instances of E, and performs all the deletion propagation actions from E listed in the preceding subsection.

$stat_{LC}(P \; \& \; Q)$ is $stat_{LC}(P); stat_{LC}(Q)$ under the assumption that $P \implies [stat_{LC}(Q)]P$.

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

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

under the assumptions $C \implies [stat_{LC}(P)]C$ and $\neg C \implies [stat_{LC}(Q)]\neg C$.

For $s{\rightarrow}forAll(x|P)$, $stat_{LC}$ is defined as $for\ x : s\ do\ stat_{LC}(P)$, when $rd(s) \cap wr(P) = \{\}$ and $rd(P) \cap wr(P) = \{\}$.

For existential quantifiers $E{\rightarrow}exists(e|P_1\ \&\ \ldots\ \&\ P_n)$, their $stat_{LC}$ effect only creates a new $e : E$ instance 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 a primary key pk and a conjunct P_i is of the form $e.pk = value$, then $stat_{LC}(E{\rightarrow}exists(e|P_1\ \&\ \ldots\ \&\ P_n))$ is

```
$var\ e : E;$
$e := createByPKE(value);$
$stat_{LC}(Pred)$
```

Where $Pred$ is $P_1\ \&\ \ldots\ \&\ P_n$ with P_i omitted. $createByPKE(val)$ looks up $E[val]$, if this is non-null it returns the instance, otherwise it creates a new E instance with primary key value val, and returns this instance.

Otherwise, $stat_{LC}(E{\rightarrow}exists(e|P_1\ \&\ \ldots\ \&\ P_n))$ has the form:

```
$var\ e : E;$
$var\ eset : Set(E);$
$eset := E.allInstances();$
$if\ eset{\rightarrow}isEmpty()$
$then$
    $e := createE();$
    $stat_{LC}(P_1\ \&\ \ldots\ \&\ P_n)$
$else$
    $(e := eset{\rightarrow}any();$
    $eset := eset{\rightarrow}select(P_1);$
    $if\ eset{\rightarrow}isEmpty()$
    $then$
        $e := createE();$
        $stat_{LC}(P_1\ \&\ \ldots\ \&\ P_n)$
    $else$
        $(e := eset{\rightarrow}any();$
        $eset := eset{\rightarrow}select(P_2);$
        $if\ eset{\rightarrow}isEmpty()$
        $then$
            $stat_{LC}(P_2\ \&\ \ldots\ \&\ P_n)$
        $else$
            $\ldots\ case\ for\ 3\ \ldots))$
```

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

```
$e := eset{\rightarrow}any();$
$eset := eset{\rightarrow}select(P_k);$
$if\ eset{\rightarrow}isEmpty()$
$then$
    $stat_{LC}(P_k\ \&\ \ldots\ \&\ P_n)$
$else$
    $\ldots\ case\ for\ k+1\ \ldots$
```

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

```
$e := eset{\rightarrow}any();$
$eset := eset{\rightarrow}select(P_n);$
$if\ eset{\rightarrow}isEmpty()$
$then$
    $stat_{LC}(P_n)$
$else$
    $e := eset{\rightarrow}any()$
```

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 2 source instances.

IV. PATTERNS FOR MODEL TRANSFORMATIONS

In [38] we identified 29 design patterns which have been used for model transformation specification and design. According to [43], some of the most widely-used MT patterns in practice are Entity Splitting, Structure Preservation, Map Objects before Links and Entity Merging. Our approach to bx specification using UML-RSDS applies to transformations τ constructed using these patterns and others, and enables the automatic derivation of inverse transformations $\tau^{\leftarrow}$ which also use the patterns, according to Table 2.

TABLE 2. Correspondence of MT patterns.

Pattern in τ	*Pattern in* $\tau^{\leftarrow}$
Entity Splitting	Entity Merging
Entity Merging	Entity Splitting
Map Objects before Links	Map Objects before Links
Structure Preservation	Structure Preservation
Phased Construction	Phased Construction
Flattening	Unflattening
Unflattening	Flattening
Inverse Recursive Descent	Inverse Recursive Descent

Here we give a more precise characterization of the patterns, and explain their specialization for use in bx. In each case in the following patterns, the predicates $TCond$, P contain no quantifiers or calls to update operations, and have an operational interpretation $stat_{LC}$. They can contain calls of purely functional operations that do not create instances or modify data.

- **Structure Preservation:** a rule in which one source entity type S_i is mapped to one target entity type T_j, with no dependency upon any other target entity types.
  Rules C_i have the form

$$\begin{aligned} & S_i :: \\ & \quad SCond_i(self) \implies \\ & \quad\quad T_j \rightarrow exists(t \mid TCond_j(t) \;\&\; P_j(self, t)) \end{aligned}$$

  where $SCond_i$ refers only to source model entity types and features. In the case of a bx, identity attributes idS_i and idT_j of the entity types can be used to correlate instances of S_i and T_j.
- **Phased Construction:** as for Structure Preservation, but with previously-created instances of target entity types $TRef$ derived from $SRef$ instances being looked up in the succedent of a rule creating an instance of target entity T_j, in order to define the value of a $TRef$-typed role rr of the T_j instance, as in Figure 2. P_j in this case contains some equation $t.rr = TRef[r.idSRef]$. For example:

$$\begin{aligned} & S_i :: \\ & \quad SCond_i(self) \implies \\ & \quad\quad T_j \rightarrow exists(t \mid t.rr = TRef[r.idSRef]) \end{aligned}$$

  This can be used for single-valued r, rr as well as for collection-valued association ends.
- **Entity Splitting:** this pattern involves one source entity type mapping to two or more different target entity types. There are two variants of this pattern: (i) *horizontal*: two or more separate rules map from S_i, with disjoint $SCond_i$ application conditions, to distinct target entity types T_k, T_l; (ii) *vertical*: a single rule maps one S_i instance to multiple linked instances of different T_k, T_l. The second case is common in refinement transformations. Constraints, eg., C_i, will have the form

$$\begin{aligned} & S_i :: \\ & \quad SCond_i(self) \implies T_k \rightarrow exists(t1 \mid \\ & \quad\quad T_l \rightarrow exists(t2 \mid TCond_1(t1) \;\& \\ & \quad\quad\quad TCond_2(t1, t2) \;\&\; P(self, t1, t2))) \end{aligned}$$

  Some explicit means of recording the semantic link between $t1$ and $t2$ is needed to support bx definition for such transformations, this link data can be based on identity attributes. This should be the only constraint in the transformation that creates T_l instances. Other constraints can create T_k instances if their $TCond$ conditions are disjoint from $TCond_1$.
- **Entity Merging:** As with Entity Splitting, there are two variants, which are inverses to the corresponding versions of Entity Splitting. In *horizontal* Entity Merging, two or more source entity types S_i, S_j are mapped to the same target entity type T_k by separate rules:

$$\begin{aligned} & S_i :: \\ & \quad SCond_i(self) \implies \\ & \quad\quad T_k \rightarrow exists(t \mid TCond_i(t) \;\&\; P_i(self, t)) \end{aligned}$$

  and

$$\begin{aligned} & S_j :: \\ & \quad SCond_j(self) \implies \\ & \quad\quad T_k \rightarrow exists(t \mid TCond_j(t) \;\&\; P_j(self, t)) \end{aligned}$$

  For bx, sufficient information must be inserted into the T_k instances to enable the source entity of the instance (S_i or S_j) to be identified. Usually $TCond_i$ and $TCond_j$ would be logically disjoint for this reason:

$$TCond_j \implies not(TCond_i)$$

  In *vertical* Entity Merging, two or more source instances are used to construct a single target instance, in a single rule:

$$\begin{aligned} & S_1 :: \\ & \quad SCond_1(self) \;\&\; s2 : S_2 \;\& \\ & \quad SCond_2(self, s2) \implies \\ & \quad\quad T \rightarrow exists(t \mid TCond(t) \;\&\; P(self, s2, t)) \end{aligned}$$

  This rule iterates over pairs $self : S_1, s2 : S_2$, potentially creating a $t : T$ instance for each pair that satisfies the antecedent conditions.
- **Map Objects before Links:** This pattern, also known as Entities before Relations [49], separates the construction of target instances from the linking of target instances based on source links (Figure 2).

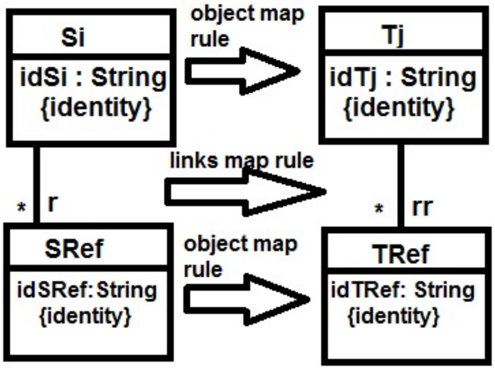


FIGURE 2. Map Objects before Links pattern.

  Entity mappings are as for any of the above patterns, whilst the linking constraints/rules have the form:

$$\begin{aligned} & S_i :: \\ & \quad T_j[idS_i].rr = TRef[r.idSRef] \end{aligned}$$

  These define target model association ends rr from source model association ends r, looking-up target

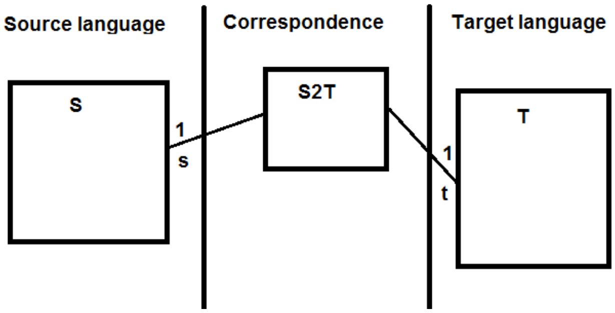


FIGURE 3. Auxiliary Correspondence Model pattern.

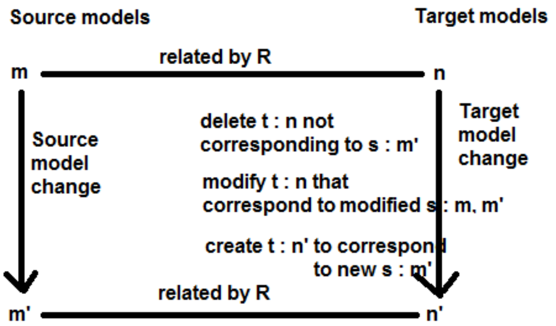

Source models
Target models
related by R
m
n
Source model change
delete t : n not corresponding to s : m'
modify t : n that correspond to modified s : m, m'
create t : n' to correspond to new s : m'
Target model change
related by R
m'
n'

FIGURE 4. Cleanup before Construct pattern.

model elements $T_j[idS_i]$ and $TRef[r.idSRef]$ which have already been created by preceding entity mapping rule(s).

The Families to Persons transformation illustrates Entity Splitting (variant (i)). In the forward direction, FamilyMember instances are mapped to Male instances if they occur in the *father* or *sons* collections of some family, and to Female instances if they occur in the *mother* or *daughters* collections. This is an example of Entity Splitting variant (i): an instance of one source entity is mapped to one instance of possibly different alternative target entities under disjoint assumptions.

We have also identified a number of specialized patterns for bx, based on inspection of published examples of bx [43]:

- **Auxiliary Correspondence Model:** using auxiliary entity types or features, maintain an explicit correspondence between source model and target model elements to facilitate change-propagation in source to target or target to source directions (Figure 3). This pattern is a built-in mechanism of the TGG language and has been used in numerous transformation cases [3], [16], [17], [21], [22], [24], [25], [27], [29], [32], [41], [55], [58], [67]. In QVT-R there are implicit trace entities for each relation [51].
- **Cleanup before Construct:** to enable change-propagation for $R^{\rightarrow}$, define additional rules that remove superfluous target model elements which are not in the bx relation R with any source elements, before constructing target elements related to source elements, and similarly for $R^{\leftarrow}$ (Figure 4). This pattern involves the introduction of additional rules, prior to the main rules of the transformation, to remove target elements which fail to correspond to source elements. Alternatively, cleanup actions can be performed after the main transformation, as with QVT-R *delete* actions [51].
- **Unique Instantiation for bx:** With this pattern, the $E \rightarrow exists(e|P)$ quantifier in rule succedents, for concrete entity type E, is interpreted as "create a new instance e of E and establish P for e, unless there already exists an $e : E$ satisfying P". This corresponds to the 'check before enforce' semantics for QVT-R [59].

  As described in Section III-C, in the case that E has a primary key eId, $E \rightarrow exists(e|e.eId = v \;\&\; P)$ is interpreted as the sequence of steps: (i) lookup $E[v]$, if this exists then assign it to e, otherwise create a new E instance with $eId = v$ and assign this to e; (ii) try to establish P for e, taking no action if P already holds.

  This avoids the deletion and recreation of elements t in one model which correspond by key to an element s in the other model, but which have divergent data. Instead the data of t is modified to enforce the transformation relation.

  If E does not have a primary key, then $stat_{LC}$ $(E \rightarrow exists(e|P))$ selects an $e : E$ for which a maximal initial part P_1 of P holds and tries to establish the remainder of P for this e, or it creates a new e and establishes P for it, if no element of E satisfies any initial part P_1 of P.
- **Lens:** the source to target mapping is independent of the target model data, with the target model $t : TL$ computed as a view or abstraction $get(s)$ containing partial information from the source model $s : SL$ [15], [6]. A dual map $put : SL \times TL \rightarrow SL$ propagates view changes to the source, such that $put(s, get(s)) = s$ and $get(put(s, t)) = t$.

  In our formalism we define view-update functions put_E for views get_E in order to reverse fine-grained data relations $P(s, t)$ such as $t.g = get_E(s.f)$ between source model data $s.f$ and target data $t.g$, for a wide range of expression forms get_E, as in [34] and [61]. The inverse $P^{\sim}(s, t)$ of $P(s, t)$ is defined as $s.f = put_E(s.f@pre, t.g)$ in such a case.

Our approach to implement Auxiliary Correspondence Model is to introduce String-valued identity attributes (primary keys) for source and target entities. These attributes are used to record which source and target instances correspond: target instance t corresponds to source s if they have the same primary key values. For example, the person to family case would be extended by this pattern as shown in Figure 5.

In addition, we introduce here two new bx patterns, which we have used in a large-scale code generation transformation (Section VII):

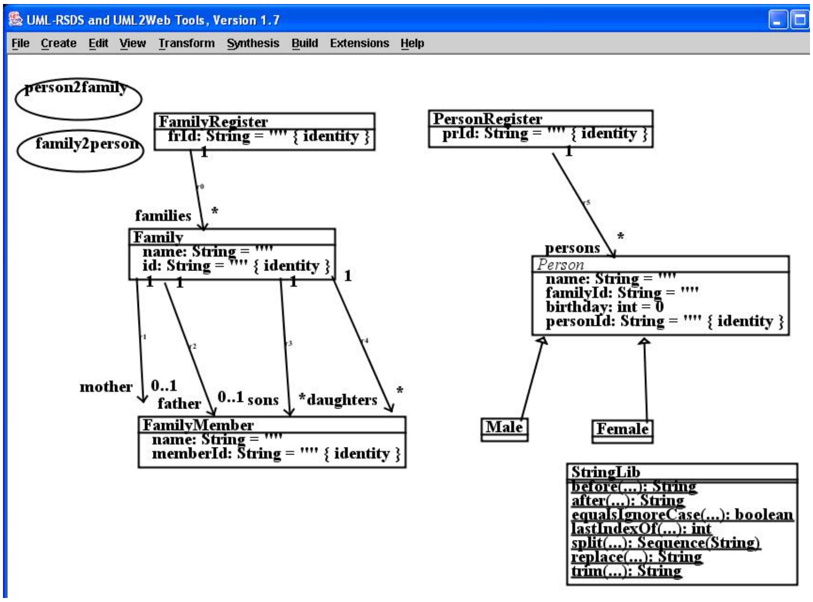


FIGURE 5. Families to Persons with Auxiliary Correspondence Model.

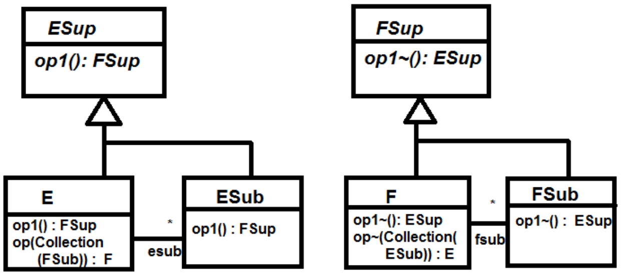


FIGURE 6. Inverse Recursive Descent pattern.

- **Inverse Recursive Descent:** if a forward transformation $\tau^{\rightarrow}$ is defined using structural recursion on the source language SL, the reverse transformation $\tau^{\leftarrow}$ may be derived from $\tau^{\rightarrow}$ as a dual structural recursion on TL (Figure 6). If source entity E has subelements $esub : Collection(ESub)$, and the corresponding target entity F has corresponding subelements $fsub : Collection(FSub)$, then a forward rule

$$\begin{aligned}
&E ::\\
&\qquad true \implies self.op1()
\end{aligned}$$

based on operations $op1$ and op of E of the form

$$\begin{aligned}
&E ::\\
&op1() : FSup\\
&pre :\ true\\
&post :\\
&\qquad result = op(esub.op1())\\
&E ::\\
&op(fs : Collection(FSub)) : F\\
&pre :\ fs = FSub[esub.eId]\\
&post :\\
&\qquad ECond \implies F{\rightarrow}exists(f\,|\,f.fId = eId\ \&\\
&\qquad\qquad FCond(f)\ \&\ f.fsub = fs\ \&\\
&\qquad\qquad P(self, f)\ \&\ result = f)
\end{aligned}$$

has the transformation invariant

$$\begin{aligned}
&F ::\\
&FCond \implies E{\rightarrow}exists(e|e.eId = fId\ \&\\
&\qquad ECond(e)\ \&\ e.esub = ESub[fsub.fId]\ \&\\
&\qquad P(e, self))
\end{aligned}$$

provided that F is only created/modified by op.
The inverse operations for the reverse transformation are:

$$\begin{aligned}
&F ::\\
&op1^{\sim}() : FSup\\
&pre :\ true\\
&post :\\
&\qquad result = op^{\sim}(fsub.op1^{\sim}())\\
&F ::\\
&op^{\sim}(es : Collection(ESub)) : E\\
&pre :\ es = ESub[fsub.fId]\\
&post :\\
&\qquad FCond \implies E{\rightarrow}exists(e|e.eId = fId\ \&\\
&\qquad\qquad ECond(e)\ \&\ e.esub = es\ \&\\
&\qquad\qquad P^{\sim}(e, self)\ \&\ result = e)
\end{aligned}$$

The operation $op1^{\sim}$ is used in an inverse rule:

$$\begin{aligned}&F ::\\ &\qquad true \implies self.op1^{\sim}()\end{aligned}$$

- **Flattening/Unflattening:** a special case of Entity Merging/Splitting where a nesting of components within a composite element is removed in the forward direction and restored in the reverse direction (Figure 7). A forward rule will have the form:

$$\begin{aligned}&E ::\\ &\quad f : fs \;\&\; FCond(f) \implies\\ &\qquad G{\rightarrow}exists(g|\\ &\qquad\qquad g.gId = f.fId \;\&\; g.eId = eId \;\&\\ &\qquad\qquad GCond(g) \;\&\; P(self, f, g))\end{aligned}$$

The corresponding invariant is

$$\begin{aligned}&G ::\\ &\quad GCond(self) \implies\\ &\qquad E{\rightarrow}exists(e|e.eId = eId \;\&\\ &\qquad\quad F{\rightarrow}exists(f|\\ &\qquad\qquad f.fId = gId \;\&\; f : e.fs \;\&\\ &\qquad\qquad FCond(f) \;\&\; P(e, f, self)))\end{aligned}$$

The Family to Persons transformation also illustrates Flattening/Unflattening, with the containment hierarchy of Family-Member within Family being flattened in the target model. The case uses an additional attribute *familyId* (playing the role of *eId* in Figure 7) to record in the target model element which source composite its corresponding source instance belongs to.

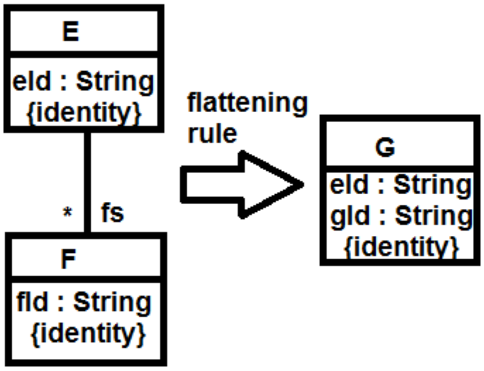


FIGURE 7. **Flattening/Unflattening pattern.**

The enhanced rules of *family2person* using the Auxiliary Correspondence Model and Flattening/Unflattening patterns are then:

```
Family::
 m : mother->union(daughters) =>
    Female->exists( p |
       p.personId = m.memberId &
       p.familyId = id &
       p.name = name + ", " + m.name )
```

```
Family::
 m : father->union(sons) =>
    Male->exists( p |
       p.personId = m.memberId &
       p.familyId = id &
       p.name = name + ", " + m.name )
```

There are corresponding derived invariants. Other forms of flattening discard certain model elements (eg., if only leaf classes in a class diagram are mapped to relational tables) and associations between elements (eg., state composition links in the state machine flattening example of [46]). The reverse transformation then needs to reconstruct the model elements or structure.

We can extend the patterns of Table 2 to include update operation calls within succedents, provided the operations are invertible in the manner of the inverse recursive descent pattern above. An operation defined as

$$\begin{aligned}&S_i ::\\ &op() : T_j\\ &post :\\ &\quad SCond_i(self) \implies\\ &\qquad T_j{\rightarrow}exists(t|TCond_j(t) \;\&\; P_j(self, t) \;\&\\ &\qquad\qquad result = t)\end{aligned}$$

can be inverted to:

$$\begin{aligned}&T_j ::\\ &op^{\sim}() : S_i\\ &post :\\ &\quad TCond_j(self) \implies\\ &\qquad S_i{\rightarrow}exists(s|SCond_i(s) \;\&\; P_j^{\sim}(s, self) \;\&\\ &\qquad\qquad result = s)\end{aligned}$$

Applications $y.rr = x.r.op()$ in a constraint succedent invert to $x.r = y.rr.op^{\sim}()$.

If T_j elements are only created/modified by op, then the transformation satisfies the invariant:

$$\begin{aligned}&T_j ::\\ &\quad TCond_j(self) \implies\\ &\qquad S_i{\rightarrow}exists(s|SCond_i(s) \;\&\; P_j(s, self))\end{aligned}$$

V. APPLICATION OF PATTERNS TO CONSTRUCT BX

A unidirectional transformation τ_0 defined using the patterns of Table 2, and consisting of type 1 constraints satisfying syntactic non-interference, can be extended by the use of the Auxiliary Correspondence Model and Cleanup before Construct patterns to a transformation $\tau^{\rightarrow}$ which is the forward part of a bx. The bx relation and reverse transformation $\tau^{\leftarrow}$ can also be derived.

An alternative condition to syntactic non-interference is *semantic non-interference*: a use case with postconditions C_1 to C_n satisfies this property if for $i < j$:

$$C_i \implies [stat_{LC}(C_j)]C_i$$

TABLE 3. Inverses of predicates and expressions.

P(s,t)	$P^{\sim}(s,t)$	Condition
$t.g = s.f$	$s.f = t.g$	Assignable features f, g
$t.g = s.f.fun$	$s.f = t.g.fun^{\sim}$	Invertible function/composition *fun*, Eg., $(cosh.log)^{\sim}$ is $exp.acosh$
$t.y = if\ C(s.x)\ then\ f(s.x)$ $\quad else\ g(s.x)\ endif$	$s.x = if\ C'(t.y)\ then\ f^{\sim}(t.y)$ $\quad else\ g^{\sim}(t.y)\ endif$	f on C, g on $not(C)$ are invertible with disjoint ranges. $C'(z)$ expresses that z is in the range of f on C.
$t.g = 1/s.f.fun$	$s.f = (1/t.g)\rightarrow fun^{\sim}()$	Invertible double-valued *fun*
$t.g = s.f\rightarrow pow(x)$	$s.f = t.g\rightarrow pow(1.0/x)$	x non-zero, independent of $s.f$
$t.g = a\rightarrow pow(s.f)$	$s.f = t.g\rightarrow log()/a\rightarrow log()$	$a > 0$, constant
$t.g = s.f.exp()$ $t.g = s.f.sin()$ $t.g = s.f.cos()$ $t.g = s.f.tan()$	$s.f = t.g.log()$ $s.f = t.g.asin()$ $s.f = t.g.acos()$ $s.f = t.g.atan()$	$t.g > 0$ $-1 \leq t.g \leq 1$ $-1 \leq t.g \leq 1$
$t.b2 = not(s.b1)$	$s.b1 = not(t.b2)$	Boolean attributes $b1$, $b2$
$t.g = K * s.f + L$ Numeric constants $K, L, K \neq 0$	$s.f = (t.g - L)/K$	f, g numeric attributes
$t.g = s.f + SK$ $t.g = SK + s.f$	$s.f = t.g.subrange(1, t.g.size - SK.size)$ $s.f = t.g.subrange(SK.size + 1, t.g.size)$	String constant SK String constant SK
$t.g = s1.f1 + SK + s2.f2$	$s1.f1 = StringLib.before(t.g, SK)$ & $s2.f2 = StringLib.after(t.g, SK)$	String constant SK not occurring in $s1.f1$, $s2.f2$
$t.g = s.f\rightarrow reverse()$	$s.f = t.g\rightarrow reverse()$	String-valued f
$t.g = StringLib.rotate(s.f, n)$	$s.f = StringLib.rotate(t.g, -n)$	String-valued f
$R(s,t)$ & $Q(s,t)$	$R^{\sim}(s,t)$ & $Q^{\sim}(s,t)$	
$s.f\rightarrow forAll(x \mid$ $\quad F\rightarrow exists(y \mid P(x,y)$ & $y : t.g))$	$t.g\rightarrow forAll(y \mid$ $\quad E\rightarrow exists(x \mid P^{\sim}(x,y)$ & $x : s.f))$	E is the entity element type of $s.f$

Syntactic non-interference implies semantic non-interference, but not conversely.

The bx derivation process consists of the following steps to enhance τ_0:

1) Introduce identity attributes in source and target entity types where necessary to support the Auxiliary Correspondence Model and Flattening/Unflattening patterns, and extend the postconditions to use these attributes, forming extended postconditions $Post_\tau$ of an enhanced transformation τ of τ_0 (cf., the step from Figure 1 to Figure 5).
2) Compute the invariant Inv_τ of this extended transformation.
3) The conjunction $Post_\tau$ & Inv_τ is the bx relation R_τ of the extended transformation (and the logical relation of the 'transformation model' of τ in the sense of [9]).
4) Further extend τ to a forward transformation $\tau^{\rightarrow}$ by applying Cleanup before Construct to add rules (applied before the main rules of τ) which remove target instances that fail to satisfy Inv_τ with respect to corresponding source instances.
5) Compute an explicit form $Post_\tau^{\sim}$ of Inv_τ, to form the basis $\tau^{\sim}$ of the reverse transformation $\tau^{\leftarrow}$. This step requires that the computations $t.g = get_E(s.f)$ of target data $t.g$ in terms of source data $s.f$ used within the postcondition predicates of τ are invertible using view update functions put_E in the sense of a lens.

For transformation postconditions conforming to the patterns of Table 2 and consisting only of type 1 constraints, using invertible computations of target data based on source data which are known as valid *get*/*put* pairs, Inv_τ and $Post_\tau^{\sim}$ can be derived mechanically from $Post_\tau$ using the definition of invariant predicates and inverse predicates $P^{\sim}$ given in Tables 3, 4, 5, 6 and in other catalogued cases. Steps 2) to 5) can therefore be automated for transformations satisfying these restrictions. Note that in the final case of Table 3 we permit a quantification of the form *forAll exists* within the P predicate of a constraint, thus extending slightly the range of constraints to which the bidirectionalization procedure can be applied.

Given a type 1 structure preservation rule C_i of the form

$$
\begin{array}{l}
S_i :: \\
\quad SCond_i(self) \implies \\
\qquad\qquad T_j\rightarrow exists(t|TCond_j(t)\ \&\ P_j(self, t))
\end{array}
$$

the corresponding invariant constraint Inv_i is:

$$
\begin{array}{l}
T_j :: \\
\quad TCond_j(self) \implies \\
\qquad\qquad S_i\rightarrow exists(s|SCond_i(s)\ \&\ P_j(s, self))
\end{array}
$$

The invariant derived from the rule expresses that each target model instance of T_j corresponds to some source model instance of S_i via that rule.

Tables 3 and 4 show some examples of inverses $P^{\sim}$ of expressions and predicates P. The computation of these inverses are implemented in the UML-RSDS tools as a higher-order transformation (the *reverse* option for use cases). Intersection on sequences is treated as *select*:

TABLE 4. Inverse of predicates on associations.

$P(s,t)$	$P^{\sim}(s,t)$	Condition
$t.rr = s.r \rightarrow reverse()$	$s.r = t.rr \rightarrow reverse()$	r, rr ordered association ends
$t.rr = s.r \rightarrow first()$	$s.r \rightarrow first() = t.rr$	r ordered association end
$t.rr = s.r \rightarrow last()$	$s.r \rightarrow last() = t.rr$	r ordered association end
$t.rr = s.r \rightarrow including(s.p)$	$s.r = t.rr \rightarrow front()$ & $s.p = t.rr \rightarrow last()$	rr, r ordered association ends p 1-multiplicity end
$t.rr = s.r \rightarrow append(s.p)$		
$t.rr = Sequence\{s.p1, s.p2\}$	$s.p1 = t.rr \rightarrow at(1)$ & $s.p2 = t.rr \rightarrow at(2)$	rr ordered association end $p1, p2$ 1-multiplicity ends
$t.rr = s.r \rightarrow select(P)$	$s.r = s.r@\mathrm{pre} \rightarrow reject(P) \cup$ $t.rr \rightarrow select(P)$	r, rr set-valued
$t.rr = s.r \rightarrow sort()$	$s.r = t.rr \rightarrow asSet()$	r set-valued, rr sequence-valued
$t.rr = s.r \rightarrow asSequence()$		
$t.rr = s.r \rightarrow sort()$	$s.r = (s.r@\mathrm{pre} \ \cap \ t.rr) \frown (t.rr - s.r@\mathrm{pre})$	r, rr sequence-valued

TABLE 5. Inverse of constraints.

Cn	$Cn^{\sim}$
S_i :: $SCond(self) \Rightarrow T_j \rightarrow exists(t \mid$ $TCond(t)$ & $P(self, t))$	T_j :: $TCond(self) \Rightarrow S_i \rightarrow exists(s \mid$ $SCond(s)$ & $P^{\sim}(s, self))$
S_i :: $v : F$ & $SCond(self, v) \ \Rightarrow$ $T_j \rightarrow exists(t \mid TCond(t)$ & $P(self, v, t))$	T_j :: $TCond(self) \Rightarrow S_i \rightarrow exists(s \mid$ $F \rightarrow exists(v \mid SCond(s, v)$ & $P^{\sim}(s, v, self)))$
S_i :: $v = e$ & $SCond(self, v) \ \Rightarrow$ $T_j \rightarrow exists(t \mid TCond(t)$ & $P(self, v, t))$	T_j :: $TCond(self) \Rightarrow$ $S_i \rightarrow exists(s \mid SCond(s, e)$ & $P^{\sim}(s, e, self))$
S_i :: $v = TRef[r.idSRef]$ & $SCond(self) \ \Rightarrow$ $T_j \rightarrow exists(t \mid TCond(t)$ & $t.rr = v$ & $P(self, t))$	T_j :: $w = SRef[rr.idTRef]$ & $TCond(self) \Rightarrow$ $S_i \rightarrow exists(s \mid SCond(s)$ & $s.r = w$ & $P^{\sim}(s, self))$

$sq1 \rightarrow intersection(sq2)$ abbreviates $sq1 \rightarrow select(x|x \ : \ sq2)$ and hence retains the order of $sq1$. In some cases the inverse predicate is based on a functional inverse to the source-to-target mapping. For example, $t.g = not(s.f)$ can be trivially inverted to $s.f \ = \ not(t.g)$, so that the view-update $s.f \ = \ put_{not}(s.f@\mathrm{pre}, t.g)$ for get_{not} defined as not does not need to take account of the previous value $s.f@\mathrm{pre}$ of $s.f$. The case of $t.rr = s.r \rightarrow select(P)$ is an example of a non-trivial view update $s.r \ = \ put_{select}(s.r@\mathrm{pre}, t.rr)$: given a value of $t.rr$, we determine an updated $s.r$ by taking the union of $s.r@\mathrm{pre} \rightarrow reject(P)$ and $t.rr \rightarrow select(P)$. The first set are those elements of $s.r$ which are unaffected by changes to $t.rr$, whilst the second set consists of possibly new elements introduced by changes to $t.rr$. View updates are discussed in more detail in Section VI.

Table 5 summarises the inverse forms of different constraint kinds involving let variable definitions $v \ = \ E$, $v = TRef[id]$ or additional quantified variables $v : F$.[2]

In terms of the framework of [59], the source-target relation R_τ associated with a UML-RSDS transformation τ is $Post_\tau$ & Inv_τ. R_τ is not necessarily bijective (eg., in the Family to Person transformation only part of the target model is significant for the bx relation, so multiple non-isomorphic target models can be related to the same source model). For unidirectional transformations τ, the design and implementation of the forward direction of τ is normally computed as $stat_{LC}(Post_\tau)$: the UML activity derived from $Post_\tau$ when interpreted procedurally [35]. However, in order to achieve the correctness and hippocraticness bx properties, Inv_τ must also be considered: before $stat_{LC}(Post_\tau)$ is applied to the source model m and target model n, n must be cleared of elements which fail to satisfy Inv_τ. This is a case of the Cleanup before Construct pattern (Section IV).

The general cleanup constraint $Cleanup_i$ derived from Inv_i is:

$$\begin{array}{l} T_j :: \\ \quad TCond_j(self) \ \& \\ \quad not(S_i \rightarrow exists(s|SCond_i(s) \ \& \ P_j(s, self))) \implies \\ \qquad\qquad\qquad self \rightarrow isDeleted() \end{array}$$

This expresses that if a target instance t does not satisfy $Inv_i(s, t)$ for any $s : S_i$, then t is deleted.

However, this rule may delete more T_j instances than are necessary: not only instances which correspond (i) to deleted S_i instances, or (ii) to S_i instances that no longer satisfy $SCond_i$, but also (iii) to S_i instances satisfying $SCond_i$, but not $P_j(s, self)$. In this last case, $self$ should not be deleted, instead $stat_{LC}(P_j)$ should be performed to re-establish P_j between the modified s and $self$. Therefore, some form of source-target tracing is required, to distinguish these cases.

[2] $let \ v : T = e \ in \ E$ is written as $v = e \implies E$ in our notation.

TABLE 6. Inverse of predicates (extended).

$P(s,t)$	$P^{\sim}(s,t)$	Condition
$t = T_j[s.idS]$	$s = S_i[t.idT]$	idS primary key of S_i idT primary key of T_j
$t.rr = TRef[s.r{\rightarrow}reverse(){\rightarrow}collect(idS)]$	$s.r = SRef[t.rr{\rightarrow}reverse(){\rightarrow}collect(idT)]$	r, rr ordered association ends
$t.rr = TRef[s.r.idSRef]$ $idSRef$ primary key of $SRef$, $idTRef$ primary key of $TRef$	$s.r = SRef[t.rr.idTRef]$	rr association end with element type $TRef$, r association end with element type $SRef$
$T_j[s.idS].rr = TRef[s.r.idSRef]$ r has element type $SRef$, rr has element type $TRef$	$S_i[t.idT].r = SRef[t.rr.idTRef]$	$idS, idSRef$ primary keys of S_i, $SRef$ $idT, idTRef$ primary keys of T_j, $TRef$
$t.g = TRef[Set\{s\}{\rightarrow}closure(r)$ $\quad{\rightarrow}unionAll(f.idSRef)]$	$s.f = SRef[t.g.idTRef -$ $\quad s.r{\rightarrow}closure(r){\rightarrow}unionAll(f.idSRef)]$	f many-valued, injective, $r : S_i —_* S_i$ acyclic $g : T_j —_* TRef, f : S_i —_* SRef$

We apply the Auxiliary Correspondence Model pattern to introduce String-valued identity attributes (primary keys) for source and target entities, to record and identify which source and target instances correspond: source instance s corresponds to target instance t when $s.sId = t.tId$. This also means that $S_i[t.tId] = s$ and $T_j[s.sId] = t$. By using the Entity Splitting and Entity Merging patterns, described in Section IV, one-many and many-one relations between source and target instances can also be recorded using this technique.

Tables 3 and 4 can be extended to include assignments to features based on object indexing lookups (Table 6).

If identity attributes have been introduced using Auxiliary Correspondence Model, the postconditions (rules) C_i of τ have the typical form

$$\begin{array}{l}
S_i :: \\
\quad SCond_i(self) \implies \\
\quad\quad T_j{\rightarrow}exists(t \mid t.tId = sId\ \& \\
\quad\quad\quad TCond_j(t)\ \&\ P_j(self, t))
\end{array}$$

and the corresponding invariant constraints Inv_i are:

$$\begin{array}{l}
T_j :: \\
\quad TCond_j(self) \implies \\
\quad\quad S_i{\rightarrow}exists(s \mid s.sId = tId\ \& \\
\quad\quad\quad SCond_i(s)\ \&\ P_j(s, self))
\end{array}$$

The cleanup constraints can then be simplified to:

$$\begin{array}{l}
T_j :: \\
\quad TCond_j(self)\ \& \\
\quad not(tId : S_i{\rightarrow}collect(sId)) \implies self{\rightarrow}isDeleted()
\end{array}$$

and

$$\begin{array}{l}
T_j :: \\
\quad TCond_j(self)\ \&\ tId : S_i{\rightarrow}collect(sId)\ \& \\
\quad not(SCond_i(S_i[tId])) \implies self{\rightarrow}isDeleted()
\end{array}$$

which correspond to cases (i) and (ii) above. This version is also potentially more efficient than the original cleanup constraint. The second case is omitted if $SCond_i$ is the default *true*. The updates of case (iii) are carried out in the $Post_\tau$ constraints. $\tau^{\rightarrow}$ is defined to have postconditions consisting of the cleanup constraints, followed by $Post_\tau$, and hence it is an extension of τ: its postconditions imply those of τ. We denote by $\tau^{\times}$ the transformation that consists of the cleanup constraints. Then the forward transformation can be decomposed as the sequential composition of $\tau^{\times}$ and the original τ: $\tau^{\rightarrow} = \tau^{\times}; \tau$.

The cleanup constraints can be further optimized by writing them in the form:

$$\begin{array}{l}
T_j :: \\
\quad TCond_j(self)\ \& \\
\quad S_i[tId]{\rightarrow}oclIsUndefined() \implies self{\rightarrow}isDeleted()
\end{array}$$

and

$$\begin{array}{l}
T_j :: \\
\quad TCond_j(self)\ \&\ not(S_i[tId]{\rightarrow}oclIsUndefined())\ \& \\
\quad not(SCond_i(S_i[tId])) \implies self{\rightarrow}isDeleted()
\end{array}$$

A single consolidated $Cleanup_i$ constraint is:

$$\begin{array}{l}
T_j :: \\
\quad TCond_j(self) \implies \\
\quad\quad if\ S_i[tId]{\rightarrow}oclIsUndefined() \\
\quad\quad then\ self{\rightarrow}isDeleted() \\
\quad\quad else\ if\ not(SCond_i(S_i[tId])) \\
\quad\quad then\ self{\rightarrow}isDeleted() \\
\quad\quad else\ true \\
\quad\quad endif\ endif
\end{array}$$

A similar analysis and formation of invariants and cleanup constraints can be carried out for the other bx pattern cases of Table 2. Details are given in the appendix.

$\tau^{\rightarrow}$ operates on both the source and target model, and propagates changes in the source model m to the target n: if elements of m are deleted, so are the corresponding elements of n. If elements of m are modified so that no rule applies to map an m element to a particular target element, then that target is deleted (this is the implicit deletion semantics of QVT-R [8], here made explicit by the $\tau^{\times}$ rules). Otherwise, if a modified source element still corresponds by identity to a valid target element, that target is updated according to the forward rules. Finally, if a rule applies to map a source element (new or modified) to a target which does not already exist, an appropriate target is created. Changes to identity attribute values are not permitted.

VI. INVERSE TRANSFORMATIONS AND VIEW UPDATES

For the reverse transformation $\tau^{\leftarrow}$, the roles of $Post_{\tau}$ and Inv_{τ} are interchanged: elements of the source model which fail to satisfy $Post_{\tau}$ with respect to some element of the target model should be deleted or modified. The postcondition of the reverse transformation $\tau^{\sim}$ is $Post_{\tau}^{\sim}$, the explicit form of Inv_{τ}. The key difference between $\tau^{\rightarrow}$ and $\tau^{\leftarrow}$ is that $Post_{\tau}^{\sim}$ may involve *view updates* [4] of the source model, using implicit changes to source elements based on target data, because τ is specified using explicit updates of the target based on the source.

If a predicate such as $t.g = s.f{\rightarrow}last()$ or $t.g = s.f{\rightarrow}select(P1)$ is inverted, the result is a predicate $s.f{\rightarrow}last() = t.g$ or $s.f{\rightarrow}select(P1) = t.g$. These are termed view updates, because they specify an update to $s.f$ based on the required value of some view, function or selection of its data. There may not be a unique way to perform such an update.

We extend the definition of $stat_{LC}$ so that the procedural interpretation $stat_{LC}(P)$ of a view update predicate P is a statement which makes P true by making minimal necessary changes to $s.f$, using suitable put_E functions which take as input the previous value $s.f@\mathrm{pre}$ of $s.f$ and the updated value $t.g$ of the target feature: $s.f := put_E(s.f@\mathrm{pre}, t.g)$. This approach can be used to implement target-to-source change propagation for a bx where the target model is constructed from views of the source model.

Tables 7, 8, 9 show the extended definitions of $stat_{LC}$ for some common view update predicates.

In the cases for *tail* and *front* in Table 7, d is the default element of the element type of f. In the case for *collect*, $e^{\sim}$ is an inverse to e, defined, for example, according to Table 3. Notice that in this case, elements are deleted from f if their e value is not in g, then any extra elements y where $y.e$ is in g but y is not in $f@\mathrm{pre}$, are added to f. $s{\rightarrow}intersection(t)$ is treated as for $s{\rightarrow}select(x|x : t)$.

An assignment $t.g = D[s.f{\rightarrow}select(P){\rightarrow}collect(bId)]$ can be inverted to $s.f{\rightarrow}select(P) = B[t.g.dId]$ if D is the corresponding target entity type for source entity B, and similarly for other cases of binary and unary collection operators in place of *select*. The view update definitions can therefore also be used for bx that involve source-target correspondences using identity attributes.

TABLE 7. View update interpretations for =.

P	$stat_{LC}(P)$

$f{\rightarrow}last() = x$	if $f.size = 0$
	then $f := Sequence\{x\}$
	else $f[f.size] := x$

$f{\rightarrow}front() = g$	if $f.size > 0$
	then $f := g \frown Sequence\{f.last\}$
	else $f := g \frown Sequence\{d\}$

$f{\rightarrow}first() = x$	if $f.size = 0$
	then $f := Sequence\{x\}$
	else $f[1] := x$

$f{\rightarrow}tail() = g$	if $f.size > 0$
	then $f := Sequence\{f.first\} \frown g$
	else $f := Sequence\{d\} \frown g$

$f{\rightarrow}select(P) = g$ sequence-valued f	$f := f{\rightarrow}select(x \mid (x : g \,\&\, P) \ or \ not(P)) \frown$ $(g - f@\mathrm{pre})$
$f{\rightarrow}reject(P) = g$	Same as $f{\rightarrow}select(not(P)) = g$
$f{\rightarrow}collect(e) = g$	$f := f@\mathrm{pre}{\rightarrow}reject(x \mid x.e \notin g) \frown$ $(g{\rightarrow}collect(e^{\sim}) - f@\mathrm{pre})$

TABLE 8. View update interpretations for :.

P	*derived predicate/*$stat_{LC}(P)$
$x : a{\rightarrow}intersection(b)$	$x : a \,\&\, x : b$
$x : a - b$	$x : a \,\&\, x \ / : \ b$
$x : a{\rightarrow}select(P)$	$x : a \,\&\, x.P$
$x : a{\rightarrow}reject(P)$	$x : a \,\&\, x.not(P)$
$x : a{\rightarrow}asSet()$	$x : a$
$x : a{\rightarrow}asSequence()$	$x : a$
$x : a{\rightarrow}sort()$	$x : a$
$x : a{\rightarrow}sortedBy(e)$	$x : a$
$x : a{\rightarrow}reverse()$	$a := a{\rightarrow}prepend(x)$

$x : a{\rightarrow}union(b)$
 	
$*$-multiplicity a
	if $a{\rightarrow}union(b){\rightarrow}includes(x)$ then skip
	else $a := a{\rightarrow}including(x)$

$x : a{\rightarrow}union(b)$
 	
0..1-multiplicity a
	if $a{\rightarrow}union(b){\rightarrow}includes(x)$ then skip
	else if $a.size = 0$
	then $a := a{\rightarrow}including(x)$
	else $b := b{\rightarrow}including(x)$

TABLE 9. View update interpretations for <:.

P	*derived predicate/*$stat_{LC}(P)$
$x <: a{\rightarrow}intersection(b)$	$x <: a \,\&\, x <: b$
$x <: a - b$	$x{\rightarrow}forAll(y \mid y : a \,\&\, y \ / : \ b)$
$x <: a{\rightarrow}select(P)$	$x{\rightarrow}forAll(y \mid y : a \,\&\, y.P)$
$x <: a{\rightarrow}reject(P)$	$x{\rightarrow}forAll(y \mid y : a \,\&\, y.not(P))$
$x <: a{\rightarrow}asSet()$	$x <: a$
$x <: a{\rightarrow}asSequence()$	$x <: a$
$x <: a{\rightarrow}sort()$	$x <: a$
$x <: a{\rightarrow}sortedBy(e)$	$x <: a$
$x <: a{\rightarrow}union(b)$	$x{\rightarrow}forAll(y \mid y : a{\rightarrow}union(b))$

An example of a case from Table 8 is the inverse mapping *person2family* of *family2person*. The $Post^{\sim}$ rules in this case have the form:

```
Female::
 Family->exists( fx | fx.id = familyId &
   FamilyMember->exists( m |
     m.memberId = personId &
     m : fx.mother->union(fx.daughters) &
```

```
    fx.name = StringLib.before(name, ", ") &
    m.name = StringLib.after(name, ", ") ) )
```

The update $m : fx.mother \rightarrow union(fx.daughters)$ adds m preferentially to $fx.mother$, only adding to $fx.daughters$ if $fx.mother$ is already set.

The cleanup constraints for $\tau^{\leftarrow}$ are as follows in the general case:

$$S_i ::$$
$$SCond_i(self)\ \&\ not(sId : T_j \rightarrow collect(tId)) \implies self \rightarrow isDeleted()$$

for each C_i, and

$$S_i ::$$
$$SCond_i(self)\ \&\ sId : T_j \rightarrow collect(tId)\ \&$$
$$not(TCond_j(T_j[sId])) \implies self \rightarrow isDeleted()$$

The latter is omitted if $TCond_j$ is not present (ie., it is *true*).

As with the forward transformation, $\tau^{\leftarrow} = \tau^{\sim\times}; \tau^{\sim}$. $stat_{LC}(Post^{\sim})$ establishes *Inv*, whilst the cleanup constraints together with $Post^{\sim}$ establish *Post*. Thus $\tau^{\leftarrow}$ satisfies correctness. Hippocraticness follows since neither the cleanup or $Post^{\sim}$ constraints modify the source model if R already holds.

View updates can be defined for certain cases of user-defined functions by using syntactically-derived view update functions as in [34], [47], and [61]. In particular, any constraint predicate $P(s, t)$ which computes a sequence-valued feature $t.g = get(s.f)$ from sequence-valued $s.f$ by using a recursively-defined function *get* on the elements of $s.f$ can be inverted to $s.f = put(s.f@\text{pre}, t.g)$ for a recursively-defined view update function *put*.

If *get* is schematically of the form:

$$\begin{aligned} get(x) = {} & if\ x.size = 0\ then\ Sequence\{\} \\ & else\ if\ P_1(x.first, x.tail)\ then \\ & \quad Sequence\{get_1(x.first)\}^\frown get(x.tail) \\ & else\ if\ \ldots \\ & else\ get(x.tail)\ endif \end{aligned}$$

Then a corresponding *put* is defined by:

$$\begin{aligned} put(x, y) = {} & if\ x.size = 0\ then\ Sequence\{\} \\ & else\ if\ P_1(x.first, x.tail)\ then \\ & \quad Sequence\{put_1(x.first, y.first)\}^\frown \\ & \qquad put(x.tail, y.tail) \\ & else\ if\ \ldots \\ & else\ put(x.tail, y.tail)\ endif \end{aligned}$$

where each put_i is an update function inverse to get_i. Similarly for recursively-defined *get* functions from trees to sequences [62].

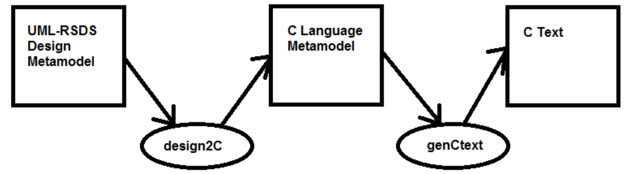


FIGURE 8. UML2C transformation architecture.

VII. CASE STUDY: MAPPING UML TO C

UML-RSDS already contains code generators for three versions of Java, and for C# and C++. If a code generator for a new language or language version is required, this can be written as a UML-RSDS transformation from the design metamodel (Figure 9 together with an OCL metamodel and activities metamodel) of UML-RSDS to the abstract and concrete syntax of the new target language, and then incorporated as a plugin into the UML-RSDS tools [44]. Here we consider the generation of ANSI C code from UML, and the extraction of a UML model from a C model. The code generator is factored into a model-to-model bx and a model-to-text transformation using the architecture of Figure 8.

The model-to-model bx *design2C* has source language the combined UML-RSDS metamodels for class diagrams, expressions and activities. Figure 9 shows the class diagram metamodel. The target language is a simplified version of the abstract syntax of C programs (Figure 10 shows the type and program structure parts of this metamodel).

We define *design2C* using the patterns of Table 2, to enable it to be used as the basis of a bx between UML and C. *design2C* consists of five subtransformations, mapping separately (i) types, (ii) class diagrams, (iii) expressions, (iv) activities, (v) use cases to C. Here we consider specifically the type mapping (i), and give extracts from (ii) and (iii).

TABLE 10. Informal mapping between UML and C.

UML element e	*C representation* e'
Class *E*	`struct E { ... };`
Property $p : T$	member `T' p;` of the struct for p's owner, where `T'` represents T

String type
int, long, double types
boolean type
	`char*`
	same-named C types
	`unsigned char`

Entity type *E*	`struct E*` type
Set(*E*) type *Sequence*(*E*) type	`struct E**` (array of E' with no duplicates) `struct E**` (array of E', possibly with duplicates)
Operation $op(p : P) : T$ of E	C operation `T' op(E' self, P' p)`

Table 10 shows the informal mapping between UML and C for classes, types and attributes.

Primitive types, entity types and collection types are successively mapped. $typeId : String$ and $ctypeId : String$ are new identity attributes introduced for Auxiliary Correspon-

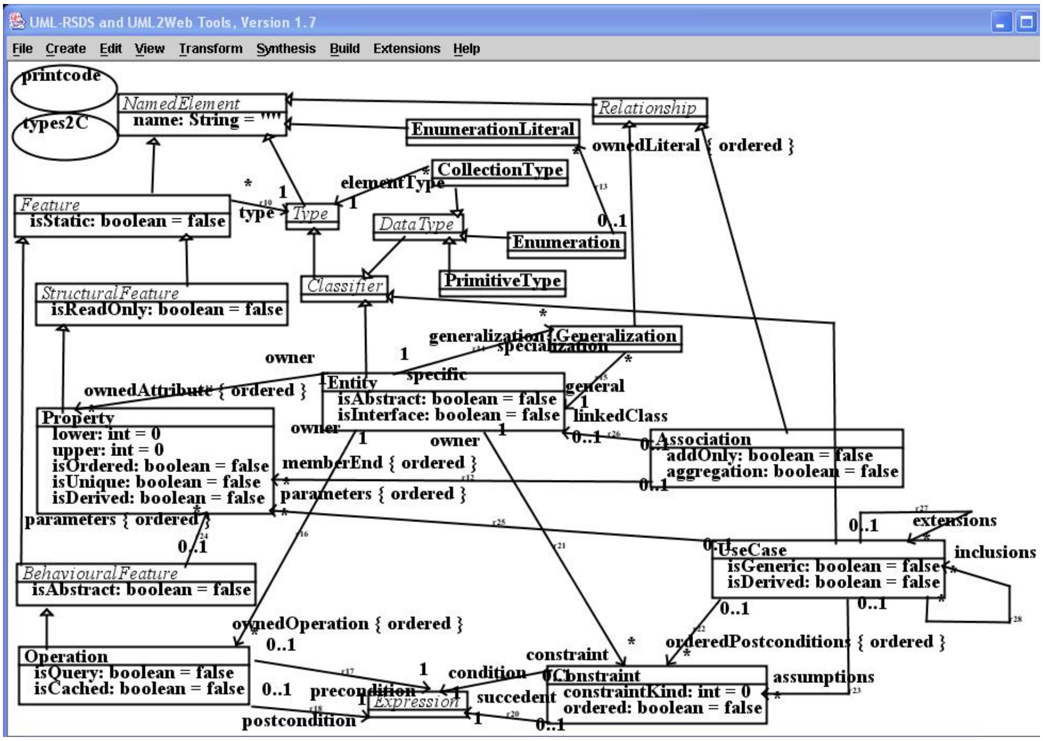


FIGURE 9. UML-RSDS class diagram metamodel.

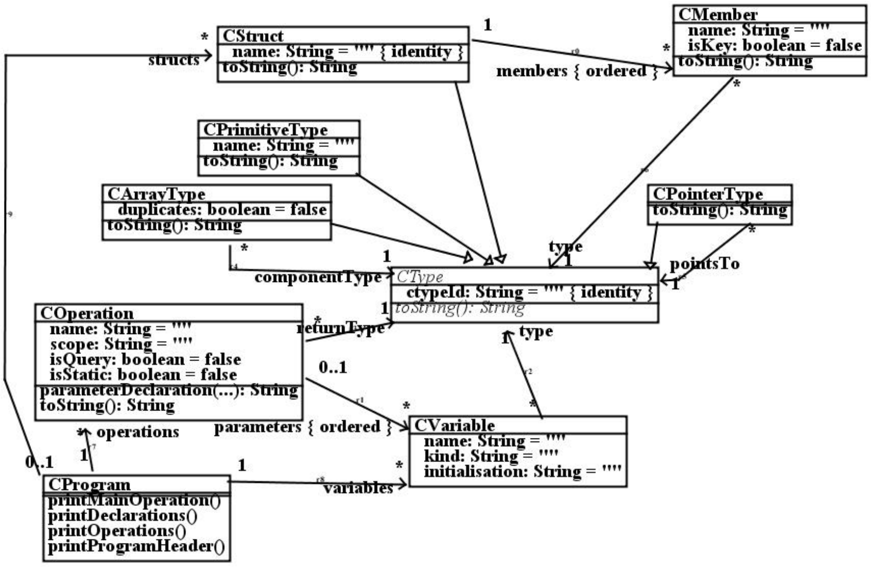


FIGURE 10. C program and types metamodel.

dence Model. These provide a correspondence between the occurrences of types in the source UML-RSDS design and in the target C implementation. *typeId* is a string consisting of digits.

The postconditions mapping UML types to C are:

```
PrimitiveType::
 name = "int"   =>
  CPrimitiveType->exists(p |
           p.ctypeId = typeId &
           p.name = "int" )
PrimitiveType::
 name = "long"   =>
  CPrimitiveType->exists(p |
           p.ctypeId = typeId &
           p.name = "long" )
PrimitiveType::
 name = "double"   =>
  CPrimitiveType->exists(p |
           p.ctypeId = typeId &
           p.name = "double" )
PrimitiveType::
 name = "boolean"  =>
  CPrimitiveType->exists( p |
           p.ctypeId = typeId &
           p.name = "unsigned char" )
PrimitiveType::
 name = "String"  =>
  CPointerType->exists( t |
     t.ctypeId = typeId &
     CPrimitiveType->exists( p |
           p.name = "char" &
           t.pointsTo = p ) )
Entity::
 CPointerType->exists( p |
   p.ctypeId = typeId &
   CStruct->exists( c |
      c.name = name &
      c.ctypeId = name & p.pointsTo = c ) )
CollectionType::
 name = "Sequence"  =>
   CArrayType->exists( a |
     a.ctypeId = typeId &
     a.duplicates = true &
     a.componentType =
          CType[elementType.typeId])
CollectionType::
  name = "Set"  =>
    CArrayType->exists( c |
     c.ctypeId = typeId &
     c.duplicates = false &
     c.componentType =
           CType[elementType.typeId])
```

For this mapping we require that collection types cannot have collection element types. Classes correspond to structs by name, this is a distinct correspondence to that of UML and C types. The char type has *ctypeId* equal to the empty string.

In subtransformation (ii), the attributes (and association ends) owned by a class are mapped to members of its corresponding struct:

```
Entity::
  CStruct->exists( c | c.name = name &
    ownedAttribute->forAll( p |
      CMember->exists( m | m.name = p.name &
          m.isKey = p.isUnique &
          m.type = CType[p.type.typeId] &
          m : c.members ) ) )
```

The *CStruct*[*name*] instance is looked-up by the *name* primary key, and is not re-created, according to the Unique Instantiation for bx pattern. The *CType* instance corresponding to *p.type* for each property *p* is looked up by its *ctypeId*.

The type mapping constraints conform to the patterns of Table 2. In particular they include an example of the Entity Merging (horizontal) pattern: *CPointerType* instances can correspond either to String type instances or to entity type instances. These cases are distinguished in the C language model, enabling a valid inverse map to be defined. The mapping of entity types is an example of (vertical) Entity Splitting: a UML class (entity type) is mapped to a C pointer type and to a C struct. In this case these target instances both belong to the same class hierarchy, so they are given distinct *ctypeId* key values, both of which are derived 1-1 from the source instance.

Following the model-to-model transformation *design*2*C*, a model-to-text transformation *genCtext* produces C text from the C metamodel. For example:

```
CStruct::
  ("struct " + name)->display() &
  "{"->display() &
  members->forAll( m |
      ("  " + m.type + "  " +
       m.name + ";")->display() ) &
  "};\n"->display()
```

A *toString*() operation is defined on each *CType* subclass to give the text representation of the C type.

Using the results of the appendix, we can show that *design*2*C* can be used as the basis for a bx. Apart from the practical benefits of change propagation in either direction, the bx property has important theoretical consequences. For high-integrity refinement transformations, the property of *conservativeness* or *model-coverage* is important [56]: that all data and information in the refined model is derived from the original model. The invariant of a refinement expresses that this property holds. Similarly, the bx relation can encode model-level semantic preservation/semantic equivalence of the source and target models: that they have equivalent semantics. Finally, the existence of change-propagation in both directions helps to support round-trip engineering between UML and C.

Transformation invariants and cleanup constraints can be derived for $design2C^{\rightarrow}$, following the definitions in Section V and the appendix. The invariants express that the C representation is derived entirely from the source UML-RSDS design, and hence conservativeness of the transformation.

The $Post^{\sim}$ constraints provide an explicit inverse (reverse-engineering or abstraction) transformation $design2C^{\leftarrow}$, and can be derived automatically:

```
CPrimitiveType::
 name = "int"  =>
   PrimitiveType->exists( t |
            t.typeId = ctypeId &
            t.name = "int" )

CPrimitiveType::
 name = "long"  =>
   PrimitiveType->exists( t |
            t.typeId = ctypeId &
            t.name = "long" )

CPrimitiveType::
 name = "double"  =>
   PrimitiveType->exists( t |
            t.typeId = ctypeId &
            t.name = "double" )

CPrimitiveType::
 name = "unsigned char"  =>
   PrimitiveType->exists( t |
           t.typeId = ctypeId &
           t.name = "boolean" )

CPointerType::
  p : CPrimitiveType & p.name = "char" &
  pointsTo = p  =>
    PrimitiveType->exists( t |
             t.typeId = ctypeId &
             t.name = "String" )

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

CArrayType::
  duplicates = true  =>
    CollectionType->exists( t |
      t.typeId = ctypeId &
      t.name = "Sequence" &
      t.elementType =
           Type[componentType.ctypeId] )

CArrayType::
  duplicates = false  =>
    CollectionType->exists( t |
      t.typeId = ctypeId &
      t.name = "Set" &
      t.elementType =
            Type[componentType.ctypeId] )
```

The inverse mapping from C members to UML properties is (from Tables 3, 6):

```
CStruct::
  Entity->exists( e | e.name = name &
    members->forAll( m |
       Property->exists( p |
         p.name = m.name &
         p.isUnique = m.isKey &
         p.type = Type[m.type.ctypeId] &
         p : e.ownedAttribute ) ) )
```

Source-to-target change propagation is supported by *design2C*$^{\rightarrow}$. For example, if a type is changed from a *Set* to a *Sequence* in a UML-RSDS model *m*, then the *Post* constraint

```
CollectionType::
 name = "Sequence"  =>
   CArrayType->exists( a |
      a.ctypeId = typeId &
      a.duplicates = true &
      a.componentType =
           CType[elementType.typeId] )
```

applies to set *duplicates* = *true* for the corresponding C type.

Other parts of the UML to C translation can also be defined as a bx in the same manner: mappings from (ii) class diagrams, (iii) expressions and (iv) activities to corresponding C elements have been defined as part of a code generator for C in the UML-RSDS tools version 1.8 (https://nms.kcl.ac.uk/kevin.lano/uml2web/). The mappings (iii) and (iv) use a Recursive Descent style of transformation, due to the self-recursive structure of expressions and activities. To invert such transformations, the Inverse Recursive Descent pattern can be used (Figure 6), based upon the Auxiliary Correspondence Model and Unique Instantiation patterns for bx, with invariants and inverse rules defined in a similar manner as for non-recursive rules. As an example, for the mapping of expressions from OCL to C, we have the following operation in the role of *op* in the pattern:

```
BinaryExpression::
mapBinaryExpression(lexp : CExpression,
          rexp : CExpression) : CExpression
pre:
  lexp = CExpression[left.expId] &
  rexp = CExpression[right.expId]
post:
  CBinaryExpression->exists( c |
         c.operator =
              Expression.cop(operator) &
         c.left = lexp & c.right = rexp &
         c.type = CType[type.typeId] &
         result = c )
```

This has the inverse *op*$^{\sim}$ from C to OCL:

```
CBinaryExpression::
mapCBinaryExpression(lexp : Expression,
          rexp : Expression) : Expression
pre:
  lexp = Expression[left.cexpId] &
  rexp = Expression[right.cexpId]
post:
  BinaryExpression->exists( e |
         e.operator =
              CExpression.uop(operator) &
         e.left = lexp & e.right = rexp &
         e.type = Type[type.ctypeId] &
         result = e )
```

where *cop* and *uop* are dual functions defining the corresponding operators in the two languages. For example, the OCL operator *and* (&) corresponds to the C operator &&.

The complete transformation consists of over 250 rules and operations, and is one of the largest UML-RSDS transformations that have been developed.

VIII. BX SPECIFICATION IN QVT-R

QVT-R [51] was designed as a declarative MT language supporting (in principle) specification of bidirectional and multi-directional transformations. However to achieve bidirectionality, various restrictions need to be placed on QVT-R specifications.

In particular, the use of recursive *where* invocations between QVT-R relations can prevent the use of a transformation as a bx [46], as can the use of 'black box' implementations of relations [51]. In addition, the semantics in the standard is incomplete and lacks formality. There is a lack of guidance for the correct organization of specifications, in particular how data-dependencies and conflicts between relations should be managed, and there is a lack of guidance on the construction of specifications that support bidirectional/multi-directional execution.

To address these problems, we have implemented a formal and extended version of the logical semantics of [51] as a mapping from QVT-R to UML-RSDS. The mapping makes explicit the semantic meaning of a QVT-R specification in a form that can be manually inspected or formally analyzed, including the use of formal proof.

A. QVT-RELATIONS (QVT-R)

QVT-R transformations are defined using *relations*, which are rules that relate elements of a source model to corresponding elements of a target model. For example the UML2C mapping from classes to C structs (and vice-versa) can be defined by:

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

Relations include at least one domain, each domain is associated to some model involved in the transformation. Domain root variables represent elements of the domain model, in this case an *Entity* instance *e* of the *design* model, and a *c* : *CStruct* instance of the *C* model. The keyword *enforce* indicates that the domain elements may be modified by the relation when executed in the direction of the model of the domain.

The relation maps an *Entity* instance to a *CStruct* instance when execution is performed in the *C* model direction. If instead the relation was executed in the *design* direction, a mapping from *CStruct* to *Entity* would result. The variable n : *String* is an auxiliary (local) variable of the relation, used to transfer feature values from one domain to the other.

Top relations correspond to use case postcondition constraints in UML-RSDS: they are applied to all source elements which satisfy their application conditions, in an arbitrary order. QVT-R also has non-top relations, which only execute if explicitly invoked from the *where* clause of a relation, and in this case the source elements are supplied as inputs to the called relation via the call parameters. Non-top relations correspond to update operations in UML-RSDS.

The UML2C mapping from classes to pointer types in QVT-R is:

```
top relation Class2CPointerType
{ enforce domain design e : Entity
    { typeId = t };
  enforce domain C p : CPointerType
    { ctypeId = t };
}
```

As with UML-RSDS rules, QVT-R relations R_j may depend upon the results of previously-executed relations R_i. In QVT-R these dependencies can be explicitly stated by a *when* $\{R_i(pars)\}$ clause in R_j. For example, the following relation creates links between existing *CPointerType* and *CStruct* instances:

```
top relation LinkCPointerTypeCStruct
{ enforce domain design e : Entity { };
  enforce domain C p : CPointerType
    { pointsTo = c : CStruct { } };
  when
  { Class2CStruct(e,c) and
    Class2CPointerType(e,p) }
}
```

This is an example of the Map objects before links pattern: the *pointsTo* link is only set once the objects *c* and *p* are available to link.

We refer to the domain root variables (*e* and *p* here) and the object template variables of their patterns (eg., *c*) as the *object variables* of the relation.

B. USING BX PATTERNS IN QVT-R

Most of the bx patterns described in Section IV can also be used in QVT-R, to define bx in the language:

- Structure Preservation: simple top relations such as *Class2CStruct* above, with no *when* clause and a single enforce domain for each model.
- Phased Construction: top relations R which construct a target element t and reference another top relation R' in their *when* clause to look up a target element t' which R links to t.
- Entity Splitting (horizontal): two or more top relations which each map the same source domain $s : S_i$ to target elements of different concrete entity types; the relations have disjoint *when* conditions; (vertical): top relations with a single source domain $s : S_i$ and multiple (> 1) target domains/variables.
- Entity Merging (horizontal): two or more top relations which map source domains of different entity types to a common target domain $t : T$, the top relations have disjoint *when* conditions; (vertical): top relations with multiple source domains/object variables and single target domains/variables.
- Map Objects before Links: a top relation does not create target elements, instead it links elements created by other relations, which are listed in its *when* clause. Eg., *LinkCPointerTypeCStruct* above.
- Lens: relation R computes a target feature $t.g$ as a *get* function of source feature(s) $s.f$. The assignment $t.g = get(s.f)$ and its inverse $s.f = put(s.f@pre, t.g)$ must both be included in the *where* clause of R.[3]
- Flattening/unflattening: hierarchical structure in the source model is discarded in the mapping to the target, possibly together with source elements. The reverse direction must reassemble the structure/elements.

The Auxiliary Correspondence Model pattern is implicitly provided by the trace mechanism of QVT-R. It can also be explicitly introduced by means of *key* declarations, specifying that designated attributes are identities. Cleanup before Construct is internally implemented by deletion actions (scheduled after creation/update actions [51]). Unique Instantiation is internally implemented by QVT-R's check-before-enforce semantics.

[3] This use of *@pre* is supported by the QVT-R to UML-RSDS translation.

The same concepts of read and write frames $rd()$, $wr()$ from UML-RSDS can be adopted for QVT-R. A significant difference to UML-RSDS is that relations have a designated application direction, with some domains considered as sources and others considered as targets. Only features and entities of target domains can be written by the relation. In addition, elements bound in the *when* clause are assumed to already exist and are not created by the relation. Eg., for *LinkCPointerTypeCStruct* above, $wr()$ in the C direction is just $\{CPointerType :: pointsTo\}$, and in the *design* direction is empty (the relation has no effect).

Relations may use auxiliary variables to transfer data between domains, eg., the string-valued variable t in *Class2CPointerType*. These are considered internal to the relation, and – as with local variables in UML-RSDS – they are not included in its write and read frame.

C. MAPPING QVT-R TO UML-RSDS

For a QVT-R transformation τ on distinct models $s : SL$ and $t : TL$ we can define corresponding unidirectional UML-RSDS transformations $\tau^{\rightarrow}$ and $\tau^{\leftarrow}$ for the two possible execution directions s to t and t to s of τ. Top relations are mapped to UML-RSDS rules, and non-top relations to UML-RSDS operations.

The enforce semantics of QVT-R ([51, Sec. B.2]) is based upon two predicates $create(R, m)$ and $delete(R, m)$ for each relation R and model m of the transformation. Enforcement of R in the m direction consists of applying $create(R, m)$ to carry out necessary creation/update actions on m required by R, and then $delete(R, m)$ to delete elements of m which are not required to exist by R.

In our mappings from QVT-R to UML-RSDS we give separate formal interpretations $Pres_\tau(m)$, $Con_\tau(m)$ and $Cleanup_\tau(m)$ for the update, creation and deletion phases of a relational transformation τ executed in the direction of model m. As in the QVT-R to QVT-Core mapping of [51], trace entities $R\$trace$ are used to record when relations R on source domains s and target domains t have been successfully applied: $R\$trace$ has properties $x : E$ for all the object variables $x : E$ of R. A tuple (a, b) for source elements a and target elements b should be in the trace $R\$trace$ iff R has been successfully applied to a to update or create b, and all these elements exist. These traces are tested when $R(a, b)$ occurs as a rule call in a *when* clause.

For relation R, $Pres_R(m)$ is a UML-RSDS constraint that defines R's change-propagation actions for elements that have already been matched (ie., by a previous execution of τ), it is defined as

$$\begin{array}{l} R\$trace :: \\ \quad whenp\ \&\ \bigwedge_{d \in sdom} cpred(d) \implies \\ \qquad\qquad\qquad \bigwedge_{d \in tdom} epred(d)\ \&\ wherep \end{array}$$

whenp is the logical interpretation of the *when* clause, *wherep* of the *where* clause, $cpred(d)$ of a non-target domain d, $epred(d)$ of a target domain d of m, and the conjunction is taken over all non-target domains *sdom* in the antecedent (ie, domains with models m', $m' \neq m$), and over target domains *tdom* in the succedent (ie, domains with model m). The scope of any *exists* quantifiers in the *epred* formulae are extended over the remainder of the succedent. Details of *whenp*, *wherep*, *cpred* and *epred* are given in [36]. We collect all $Pres_R(m)$ constraints for both top and non-top R into a UML-RSDS use case $Pres_\tau(m)$ representing a first transformation phase $Pres_\tau(m)$ of τ executed in the m direction. This phase propagates element feature changes from source models to m.

For example, $Pres_{Class2CStruct}(C)$ is

$$\begin{array}{l} Class2CStruct\$trace :: \\ \quad n = e.name \implies \\ \qquad\qquad c.name = n\ \&\ c.ctypeId = n \end{array}$$

This constraint propagates name changes from *Entity* instances to *CStruct*s.

$Con_R(m)$ deals with the case of creation of new m elements and R traces, for top relations R. It only applies if the source elements $s_1, \ldots$ are not already in $R\$trace$:

$$\begin{array}{l} :: \\ \quad whenp\ \&\ \bigwedge_{d \in sdom} cpred(d)\ \& \\ \quad not(R\$trace \rightarrow exists(tr \mid \bigwedge_i tr.s_i = s_i)) \implies \\ \qquad\qquad \bigwedge_{d \in tdom} epred(d)\ \&\ wherepx \end{array}$$

wherepx includes the creation $R\$trace \rightarrow exists(tr \mid \bigwedge_i tr.s_i = s_i\ \&\ \bigwedge_j tr.t_j = t_j)$ of a new trace. Again, *exists* quantifiers in the succedent apply over all of the succedent following their introduction. A predicate $\theta_R(m)$ is formed as for $Con_R(m)$ but without the predicates on $R\$trace$. The $Con_R(m)$ constraints are placed in a use case representing a second τ phase, $Con_\tau(m)$. These constraints propagate element creation from source models to m.

For example, $Con_{Class2CStruct}(C)$ is:

$$\begin{array}{l} :: \\ \quad e : Entity\ \&\ n = e.name\ \& \\ \quad not(Class2CStruct\$trace \rightarrow exists(tr \mid tr.e = e)) \implies \\ \qquad CStruct \rightarrow exists(c \mid c.name = n\ \& \\ \qquad\quad c.ctypeId = n\ \& \\ \qquad\quad Class2CStruct\$trace \rightarrow exists(tr \mid \\ \qquad\qquad tr.e = e\ \& \\ \qquad\qquad tr.c = c)) \end{array}$$

Unique instantiation/least change semantics is used for the operational interpretation of the *exists* quantifier: if $x : X$ already exists such that P holds for x, then $X \rightarrow exists(x \mid P)$ in a succedent does not recreate x. Target elements identified by a key value are looked-up by that value and re-used if they already exist. We use the $stat_{LC}(P)$ semantics of Section III-C for the least-change procedural interpretation of predicates P.

The $Cleanup_\tau(m)$ phase of τ removes any spurious target elements which have not been produced by any relation. For target entity E of m, $Cleanup_E$ is defined as:

$$\begin{array}{l} E :: \\ \quad not(r1\$trace\rightarrow exists(r1 | r1.e1 = self))\ \&\ \ldots\ \& \\ \quad not(rm\$trace\rightarrow exists(rm|rm.em = self)) \implies \\ \qquad\qquad self\rightarrow isDeleted() \end{array}$$

where the rk, $k = 1$ to m, are all the relations (top or non-top) in which the entity E occurs as the type of some target domain or target object template $ek : E$. All $Cleanup$ constraints are placed in a final τ phase, $Cleanup_\tau(m)$. Unlike in UML-RSDS, where the cleanup constraints precede the main transformation, in QVT-R the cleanup actions must follow the $Con_\tau(m)$ phase because traces are explicitly constructed by that phase, instead of being implicit based on identity attribute values.

For example, $Cleanup_{CStruct}$ is:

$$\begin{array}{l} CStruct :: \\ \quad not(Class2CStruct\$trace\rightarrow exists(tr| \\ \qquad tr.c = self))\ \& \\ \quad not(LinkCPointerTypeCStruct\$trace\rightarrow exists(tr| \\ \qquad tr.c = self)) \implies \\ \qquad\qquad self\rightarrow isDeleted() \end{array}$$

The $Cleanup$ constraints propagate element deletion from source models to m, because if any element of a trace object is deleted, so is the trace.

Additionally, non-top rules R are interpreted as operations with postconditions defined as for $Con_R(m)$. Details are in [36]. Regarding the $overrides$ clause, we follow [64] in removing this by expanding out the definition of an overridden relation to include an additional $when$ predicate which expresses that no overriding relation is applicable. A $when$ test on an overridden abstract relation $R(pars)$ is expanded to be a disjunction $R_1(pars)\ or\ \ldots\ or\ R_n(pars)$ on all the concrete relations R_i overriding R.

For each execution direction, the three phases are executed in the order $Pres_\tau(m)$; $Con_\tau(m)$; $Cleanup_\tau(m)$, and together should establish the following as postconditions of τ: (i) that any existing target object $tx : E$ of m must appear as a target property of some trace: $E\rightarrow forAll(tx|r1\$trace\rightarrow exists(r1|r1.t1 = tx)\ or\ \ldots\ or\ rm\$trace\rightarrow exists(rm|rm.tm = tx))$ where the ri are as for $Cleanup_E$; (ii) that all trace objects in $R\$trace$ satisfy the logical relation $\theta_R(m)$ defined by R (ie., $Pres_R(m)$ holds); (iii) for top relations R, that any tuple of source elements that satisfy the antecedent φ_R of $\theta_R(m)$ appears in some $R\$trace$.

The following restrictions are needed on the QVT-R specification τ in order that (i), (ii), (iii) are established by its logical interpretation: (a) No target features/entity names occur in the $when$ clause or source domain patterns of any relation; (b) If two relations both write to the same target features or entities, these updates are non-conflicting (ie, they create/update disjoint sets of elements of the same target entity, or update disjoint sets of features of instances of the entity); (c) there are no self-conflicts of a relation R, ie., one application of R cannot invalidate a different application; (d) for any mandatory target model association $A\ {}^{m1}\!\!\mathop{—}\limits_{r}^{1} B$, the $a.r$ element for $a : A$ must be created/linked to a by τ if a is created by τ.

Assuming (a), (b) and (c), the $Pres_\tau(m)$ phase establishes invariant (ii), since by (a) no target features/entities occur on the LHS of a $Pres$ constraint, and by (b) and (c) the effect of one $Pres$ constraint application is not invalidated by any other. Likewise, the $Con_\tau(m)$ phase establishes (iii) at its termination and preserves (ii), if there is no conflict between relation applications (of the same or different relations). For separate-models transformations, the $Cleanup$ constraints establish (i) and do not invalidate (ii) or (iii) because they only update the target model, not the source or traces. Condition (d) ensures that deletion propagation from a whole to a part object does not delete elements that are in traces (if the part is in a trace, so must the whole be). Thus by (a) and (d) $Cleanup_\tau(m)$ does not affect the truth of any Con or $Pres$ constraint for target model m.

If conditions (i), (ii) and (iii) hold, then neither $Pres_\tau(m)$, $Con_\tau(m)$ or $Cleanup_\tau(m)$ have any effect. Therefore τ directed at m satisfies the bx conditions of correctness and hippocraticness with respect to (i), (ii) and (iii) as a definition of the transformation consistency relation R between source and target models (Section I).

It is possible for τ to satisfy the correctness conditions (a) to (d) in one execution direction but not in others. For a bx, we require that the conditions hold in all execution directions.

The following design guidelines for QVT-R should also be observed: (e) Use $when$ dependencies between top-level relations where possible, in preference to $where$ invocations of relations; (f) The $when$ dependencies should be acyclic and reflect data-dependency relations between relations. Map Objects Before Links can be used to remove circular dependencies; (g) Relations should be listed in order of their dependencies so that later relations depend only on earlier ones; (h) There should be no $where$ clauses invoking top relations, and no negated relation calls in $when$ clauses.

If these conditions and the previous restrictions (a) to (d) are satisfied for a separate-models transformation τ, then the constraints of $Con_\tau(m)$ and $Pres_\tau(m)$ can be implemented as bounded loops over their source elements, simplifying verification and potentially improving efficiency.

While the above semantics follows closely that given in [51], it is not optimally efficient. Further improvements in efficiency can be made by using traces to lookup target elements based on source elements, as in UML-RSDS and ATL, instead of performing iterations through all traces.

We have automated this optimised mapping from QVT-R to UML-RSDS as a facility to import QVT-R transformations in UML-RSDS version 1.8.

IX. EVALUATION

In this section we evaluate the efficiency of the bx produced by our two approaches, using the list reversal example of [63], identify differences in the expressiveness of the approaches, and summarise the advantages and disadvantages of the approaches.

The list reversal case is used by [63] to compare different implementation approaches for QVT-R. Figure 11 shows the metamodels of the case.

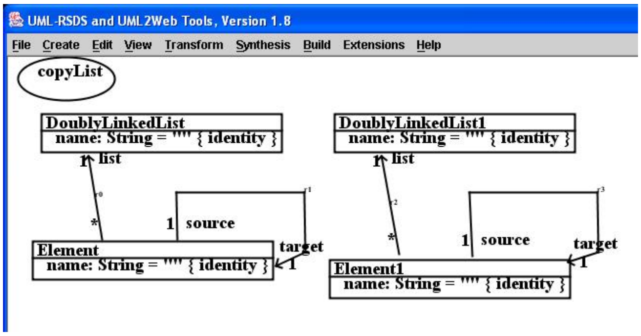

UML-RSDS and UML2Web Tools, Version 1.8
File
Create
Edit
View
Transform
Synthesis
Build
Extensions
Help
copyList
DoublyLinkedList
name: String = "" { identity }
1
list
*
1
source
target
1
Element
name: String = "" { identity }
DoublyLinkedList1
name: String = "" { identity }
1
list
*
1
source
target
1
Element1
name: String = "" { identity }

FIGURE 11. List reversal metamodels.

In the UML-RSDS version of this case study the *name* attributes of the list and element classes are assumed to be identity attributes. The forward rules are:

$$DoublyLinkedList ::$$

$$DoublyLinkedList1 \rightarrow exists(l \mid l.name = name)$$

$$Element ::$$

$$Element1 \rightarrow exists(e \mid e.name = name \;\&\; e.list = DoublyLinkedList1[list.name])$$

$$Element ::$$

$$e = Element1[name] \implies e.target = Element1[source.name]$$

This uses Structure preservation, Phased construction and Map objects before links, and is automatically invertible according to Tables 5, 6.

In our QVT-R version, Map objects before links is used to remove dependence of *element2element* on itself:

```
top relation list2list
{ enforce domain forward
    flist : DoublyLinkedList
    { name = n };
  enforce domain reverse
    rlist : DoublyLinkedList1
    { name = n };
}

top relation element2element
{ enforce domain forward
    fElement : Element
    { list = fList : DoublyLinkedList {},
      name = n };
  enforce domain reverse
    rElement : Element1
    { list = rList : DoublyLinkedList1{},
      name = n };
  when { list2list(fList,rList) }
}

top relation link2link
{ enforce domain forward
    fElement : Element
    { target = fTarget : Element {} };
  enforce domain reverse
    rElement : Element1
    { source = rSource : Element1 {} };
  when
  { element2element(fElement,rElement)
    and
    element2element(fTarget, rSource) }
}
```

The QVT-R version does not need to assume that *name* attributes are identities for tracing purposes, but instead this restriction is needed in order to ensure absence of self-conflicts in the relations (if two different elements in the source list had the same names they would be mapped to a single target element by *element2element*, and *link2link* would then attempt to set conflicting links for this element).

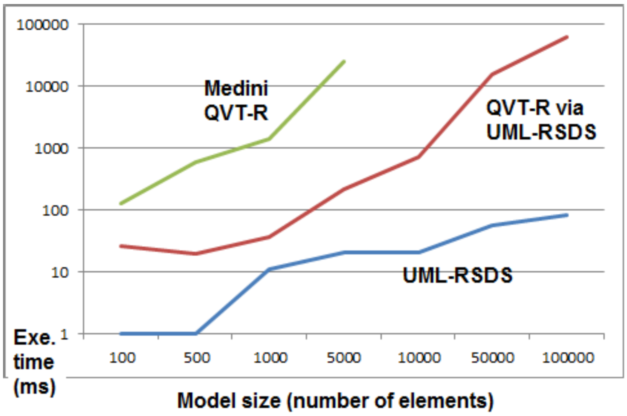


FIGURE 12. Efficiency of QVT-R versus UML-RSDS.

Figure 12 shows the execution time in ms of the UML-RSDS version and the QVT-R (translated via UML-RSDS) version, using generated Java 4 code on a 32-bit JVM and single core of a 2.53GHz Intel i3 Windows 7 laptop. The UML-RSDS version has a low time complexity, and is comparable to the performance of optimised versions of the case presented in [63]. The QVT-R via UML-RSDS version is less efficient, because of its more complex trace management. We also show the execution time using Medini QVT (projects.ikv.de/qvt) to execute the QVT-R version. This uses a 32-bit JVM on a 2.6GHz Intel i7 machine. The execution time is substantially higher than the QVT-R via UML-RSDS implementation approach.

The expressiveness of UML-RSDS and QVT-R is similar, when restricted to follow bx pattern structures and the guidelines (a) to (h) above for QVT-R. The QVT-R restriction (a) corresponds to a UML-RSDS constraint being type 1. Restriction (b) is semantic non-interference in a UML-RSDS transformation. Restriction (c) is internal consistency and confluence for a UML-RSDS rule.

A structure-preservation constraint C_i using Auxiliary Correspondence Model

$$\begin{array}{l} S_i :: \\ \quad SCond_i(self) \implies \\ \qquad T_j{\rightarrow}exists(t \mid t.tId = sId \ \& \\ \qquad\qquad TCond_j(t) \ \& \ P_j(self, t)) \end{array}$$

in UML-RSDS corresponds to a QVT-R relation:

```
top relation mapSiToTj
{ enforce domain src s : Si
    { sId = id }{ SCondi(s) };
  enforce domain trg t : Tj
    { tId = id }{ TCondj(t) };
  where { Pj(s,t) }
}
```

executed in the *trg* direction. This is a structure preservation relation in QVT-R.

A difference in expressiveness in this case is that the *where* clause in our extended QVT-R can also contain the dual predicate $P_j^{\sim}(s, t)$. This is not possible in UML-RSDS because there is no restriction of constraint write-frames based on execution direction.

A phased construction constraint has the same structure but $P_j(s, t)$ has the form $PCond_j(s, t) \ \& \ t.rr = TRef[s.r.idSRef]$ to assign a value to an association role end $t.rr$ based on prior mappings of *SRef* and *TRef* elements. In QVT-R this becomes:

```
top relation mapSiToTj
{ enforce domain src s : Si
  { sId = id, r = sref : SRef {} }
  { SCondi(s) };
  enforce domain trg t : Tj
  { tId = id, rr = tref : TRef {} }
  { TCondj(t) };
  when { mapSRefToTRef(sref,tref) }
  where { PCondj(s,t) }
}
```

This is a phased construction relation in QVT-R. Note that the mapping from r to rr is done element-by-element instead of by a single assignment as in UML-RSDS.

An entity-splitting (vertical) constraint

$$\begin{array}{l} S_i :: \\ \quad SCond_1(self) \implies \\ \qquad T_k{\rightarrow}exists(t1 \mid t1.t1Id = sId \ \& \\ \qquad\quad T_l{\rightarrow}exists(t2 \mid \\ \qquad\qquad t2.t2Id = sId \ \& \ TCond_1(t1) \ \& \\ \qquad\qquad TCond_2(t1, t2) \ \& \ P(self, t1, t2))) \end{array}$$

corresponds to an entity-splitting (vertical) relation:

```
top relation mapSiToTkTl
{ enforce domain src s : Si
    { sId = id }{ SCond1(s) };
  enforce domain trg t1 : Tk
    { t1Id = id }{ TCond1(t1) };
  enforce domain trg t2 : Tl
    { t2Id = id }{ TCond2(t1,t2) };
  where { P(s,t1,t2) }
}
```

Likewise for horizontal splitting and for entity merging constaints.

A flattening constraint

$$\begin{array}{l} E :: \\ \quad f : fs \ \& \ FCond(f) \implies \\ \qquad G{\rightarrow}exists(g \mid \\ \qquad\qquad g.gId = f.fId \ \& \ g.eId = eId \ \& \\ \qquad\qquad GCond(g) \ \& \ P(self, f, g)) \end{array}$$

corresponds to a flattening relation in QVT-R:

```
top relation mapEToG
{ enforce domain src e : E
  { eId = id, fs = f : F { fId = fid } }
    { FCond(f) };
  enforce domain trg g : G
  { gId = fid, eId = id }{ GCond(g) };
  where { P(e,f,g) }
}
```

A map objects before links linking constraint

$$\begin{array}{l} S_i :: \\ \quad T_j[idS_i].rr = TRef[r.idSRef] \end{array}$$

corresponds to a linking relation in QVT-R:

```
top relation linkTjTRef
{ enforce domain src s : Si
  { r = sref : SRef { idSRef = idref }};
  enforce domain trg t : Tj
  { rr = tref : TRef { idTRef = idref }};
  when
  { mapSiToTj(s,t) and
    mapSRefToTRef(sref,tref)
  }
}
```

mapSiToTj is a preceding rule that maps S_i to T_j, and *mapSRefToTRef* a preceding rule that maps *SRef* to *TRef*. In the QVT-R version the linking occurs element-by-element (if r and rr are collection-valued), whilst in UML-RSDS the expression $TRef[r.idSRef]$ computes in one step the collection of all *TRef* elements linked to any $self.r$ element.

Operations called by UML-RSDS rules can be mapped to QVT-R non-top relations (if they involve element creation/modification) or to QVT-R query functions (if they are purely functional). Overall, conditions (e) to (h) for the resulting QVT-R transformation follow by construction and from syntactic non-interference of the UML-RSDS transformation.

The inverse recursive descent pattern is not needed in QVT-R because the same effect can be achieved using rule

TABLE 11. Comparison of UML-RSDS and QVT-R approaches.

Approach	*Advantages*	*Disadvantages*
UML-RSDS	Efficiency. Concise. Cleanup can be performed before main transformation.	Needs identity attributes for trace. Reverse transformations are derived from forward. Inheritance only available for operations, not rules.
QVT-R	Single text for forward and reverse transformations. Traces are implicit. Rules can be inherited.	Relatively inefficient. Cleanup actions must follow main transformation.

inheritance. We can conclude that the expressiveness of the approaches is similar when restricted to the bx patterns and guidelines, and that usages of patterns in UML-RSDS translate into usages of the same patterns in QVT-R.

Table 11 compares the advantages and disadvantages of the two approaches we have presented in this paper. Whilst the UML-RSDS approach has advantages of efficiency and conciseness, the reverse transformation is derived as a secondary construct from the forward transformation. In QVT-R a single specification text describes both directions. In the Lens case this enables the specifier to write both *put* and *get* functions together, and to create new function pairs as necessary, whilst in UML-RSDS the *put* is looked-up in a fixed catalogue of *put*/*get* pairs.

X. CONCLUSION

We have defined a declarative approach for defining bidirectional transformations in UML, based on use cases with dual postcondition and invariant relations between source and target models. These use cases permit many-to-one and one-to-many bx, in addition to bijections. The definition of bx and their analysis is supported by the UML-RSDS tools [42].

By using standard UML notations and concepts, our approach should enable bx specifications to be retained and maintained independently of particular transformation technologies. The use of OCL for transformation specification facilitates the automated derivation of transformation invariants and reverse rules. Translations from ATL and ETL to UML-RSDS have been developed [37], [40], and these potentially enable our bx approach to also be applied to unidirectional transformations in these languages. Our approach scales up to transformations of practical size, and we have given extracts from a UML to C code generator with over 250 mapping rules and operations.

We have also described an approach using QVT-R, utilising a translation from QVT-R to UML-RSDS to extend the expressiveness of QVT-R, and to provide efficient execution of different directions of the QVT-R specifications. We have described how bx patterns can be used in QVT-R to support the definition of bx in the language.

APPENDIX

CORRECTNESS AND HIPPOCRATICNESS PROPERTIES FOR $\tau^{\rightarrow}$

In this section we give a formal justification that bx defined using the UML specification approach described in Section V do satisfy the bx properties of correctness and Hippocraticness, and that they satisfy local Hippocraticness and least change principles subject to some restrictions.

We consider the case of UML-RSDS transformations τ that satisfy the patterns of Table 2, with source language entities S_i and target language entities T_j, and whose constraints are of type 1, satisfying syntactic non-interference. Identity attributes are used to implement Auxiliary Correspondence Model. This means that τ is a separate-models transformation with source language SL and target language TL, and postcondition $Post_\tau$ is an ordered conjunction of constraints C_i each of the form:

$$\begin{array}{l} S_i :: \\ \quad SCond_i(self) \implies \\ \qquad T_j{\rightarrow}exists(t \mid t.tId = sId \;\& \\ \qquad\quad TCond_j(t) \;\&\; P_j(self, t)) \end{array}$$

Inv_τ is therefore a conjunction of constraints Inv_i of the form

$$\begin{array}{l} T_j :: \\ \quad TCond_j(self) \implies \\ \qquad S_i{\rightarrow}exists(s \mid tId = s.sId \;\& \\ \qquad\quad SCond_i(s) \;\&\; P_j(s, self)) \end{array}$$

A consequence is that $S_i{\rightarrow}select(SCond_i)$ and $T_j{\rightarrow}select(TCond_j)$ are isomorphic, with the isomorphism given by mapping $s : S_i$ to $T_j[s.sId]$.

The simplified cleanup constraints for $\tau^{\rightarrow}$ are therefore:

$$\begin{array}{l} T_j :: \\ \quad TCond_j(self) \;\&\; S_i[tId]{\rightarrow}oclIsUndefined() \implies \\ \qquad\qquad\qquad self{\rightarrow}isDeleted() \end{array}$$

and

$$\begin{array}{l} T_j :: \\ \quad TCond_j(self) \;\&\; tId : S_i{\rightarrow}collect(sId) \;\& \\ \quad not(SCond_i(S_i[tId])) \implies self{\rightarrow}isDeleted() \end{array}$$

The second constraint is omitted if $SCond_i$ is absent (ie., *true*). We denote the collection of the cleanup constraints by *Cleanup*. These are the postconditions of $\tau^{\times}$.

Bx correctness holds, since the cleanup constraints together with $Post_\tau$ establish Inv_τ: the *Cleanup* constraints remove any target model elements that fail to correspond by identity to source model elements in the domain of τ, and $Post_\tau$ modifies target model elements that do correspond to valid source elements, in order to re-establish Inv_τ. By construction, $stat_{LC}(Post_\tau)$ preserves Inv_τ and establishes $Post_\tau$. Therefore, the sequential composition of

$stat_{LC}(Cleanup)$ and $stat_{LC}(Post_\tau)$ establishes the bx relation R as $Post_\tau$ & Inv_τ.

Hippocraticness holds, since $stat_{LC}(Cleanup)$ has no effect if the models already satisfy R, and neither does $stat_{LC}(Post_\tau)$, due to the Unique Instantiation interpretation of the *exists* operator. Local hippocraticness holds in the sense that if corresponding elements $s : S_i$ and $t = T_i[s.sId]$ already satisfy C_i and Inv_i, then their attribute values are not modified by $\tau^{\rightarrow}$. t may however be deleted as a consequence of target element deletions from cleanup actions of other rules, and to avoid this we need the condition on TL that T_i is not subject to deletion propagation from any entity type in $wr(\tau)$. These restrictions are also assumed in the following cases.

The principle of least change is satisfied, since changes to the source model either lead to deletion of the target elements (and their incident links) which cannot be modified to correspond to source elements (cases (i) and (ii) in Section V), and deletion of any other elements required by deletion propagation, or to creation/modification of target elements, using $stat_{LC}(C_i)$. But $stat_{LC}(C_i)$ is designed as a minimally-intrusive update to the target model which will establish C_i. Interpreting a quantifier $T_j{\rightarrow}exists(t|P)$ using Unique Instantiation means that un-necessary creation of target elements is avoided.

A more complex case is Entity Merging (Section IV). In the horizontal version of this pattern two or more source entity types S_i, S_j may map to the same target entity type T_k:

$$\begin{aligned}&S_i ::\\ &\quad SCond_i(self) \implies\\ &\qquad T_k{\rightarrow}exists(t|t.tId = s1Id\ \&\\ &\qquad\quad TCond_i(t)\ \&\ P_i(self, t))\end{aligned}$$

and

$$\begin{aligned}&S_j ::\\ &\quad SCond_j(self) \implies\\ &\qquad T_k{\rightarrow}exists(t|t.tId = s2Id\ \&\\ &\qquad\quad TCond_j(t)\ \&\ P_j(self, t))\end{aligned}$$

Provided that the respective $TCond_i$, $TCond_j$ conditions are pairwise disjoint, and the sets of identities of $S_i{\rightarrow}select(SCond_i)$ and $S_j{\rightarrow}select(SCond_j)$ are disjoint, the same construction and argument for the bx correctness apply as in the Structure Preservation case. The invariants are

$$\begin{aligned}&T_k ::\\ &\quad TCond_i(self) \implies\\ &\qquad S_i{\rightarrow}exists(s|s.s1Id = tId\ \&\\ &\qquad\quad SCond_i(s)\ \&\ P_i(s, self))\end{aligned}$$

and

$$\begin{aligned}&T_k ::\\ &\quad TCond_j(self) \implies\\ &\qquad S_j{\rightarrow}exists(s|s.s2Id = tId\ \&\\ &\qquad\quad SCond_j(s)\ \&\ P_j(s, self))\end{aligned}$$

and hence $S_i{\rightarrow}select(SCond_i)$ is isomorphic to $T_k{\rightarrow}select(TCond_i)$ and $S_j{\rightarrow}select(SCond_j)$ is isomorphic to $T_k{\rightarrow}select(TCond_j)$. Semantic non-interference of $Post_\tau$ holds, because the set of T_k instances produced from S_i is disjoint from the set produced from S_j, although syntactic non-interference fails. Cleanup constraints can be derived from the invariants as for the previous cases. The vertical variant is treated similarly.

Entity Splitting (Section IV) involves one source entity type mapping to two or more different target entity types. As for entity merging, there are two versions of this pattern: (i) two or more separate postconditions on S_i, with disjoint $SCond_i$ conditions, mapping to distinct target entity types T_k, T_l; (ii) a single constraint which maps one S_i instance to multiple linked instances of different T_j. The first case is the reverse of Entity Merging (horizontal), and can be treated in a similar way to the Structure Preservation case, provided that syntactic or semantic non-interference of $Post_\tau$ holds. The second case is common in refinement transformations, and in some migrations. Constraints, eg., C_1, will have the form

$$\begin{aligned}&S_1 ::\\ &\quad SCond_1(self) \implies T_1{\rightarrow}exists(t1|t1.t1Id = sId\ \&\\ &\qquad T_2{\rightarrow}exists(t2|t2.t2Id = sId\ \&\ TCond_1(t1)\ \&\\ &\qquad\quad TCond_2(t1, t2)\ \&\ P_1(self, t1, t2)))\end{aligned}$$

Both $t1$ and $t2$ are given id values $self.sId$ (in the case that T_1 and T_2 are not disjoint, distinct identifier values derived 1-1 from source instances should be used instead). This should be the only constraint in the transformation that creates T_2 instances. Other constraints can create T_1 instances if their $TCond$ conditions are disjoint from $TCond_1$.

Inv_1 is of the form

$$\begin{aligned}&T_1 ::\\ &\quad TCond_1(self)\ \&\ t2 : T_2\ \&\ t2.t2Id = t1Id\ \&\\ &\quad TCond_2(self, t2) \implies\\ &\qquad S_1{\rightarrow}exists(s|t1Id = s.sId\ \&\ SCond_1(s)\ \&\\ &\qquad\quad P_1(s, self, t2))\end{aligned}$$

The $t2 : T_2$ expression acts like an additional $T_2{\rightarrow}forAll(t2|\ldots)$ quantifier over the remainder of the constraint. The conjunction $t2 : T_2\ \&\ t2.t2Id = t1Id$ can be optimised to $t2 = T_2[t1Id]$. For this type of rule, $S_1{\rightarrow}select(SCond_1)$ is isomorphic to $T_1{\rightarrow}select(TCond_1)$, however parts of the data of elements $s : S_1$ may be represented in the $t2 : T_2$ instances linked to the $t1 : T_1$ instance corresponding to s, rather than directly in $t1$.

The cleanup constraints for such constraints are:

$$\begin{aligned}&T_1 ::\\ &\quad TCond_1(self)\ \&\ t2 = T_2[t1Id]\ \&\\ &\quad TCond_2(self, t2)\ \&\\ &\quad not(S_1{\rightarrow}exists(s|t1Id = s.sId\ \&\ SCond_1(s))) \implies\\ &\qquad self{\rightarrow}isDeleted()\ \&\ t2{\rightarrow}isDeleted()\end{aligned}$$

The effect of such a bx is that a 'cluster' of target instances (such as $t1$, $t2$) are related to each source instance, with all elements of the cluster having identities based on the source instance identity value. An example in UML2C is the mapping of a UML class to a *CStruct* and a *CPointerType*.

Again, $stat_{LC}(Post_\tau)$ establishes $Post_\tau$ and preserves Inv_τ, whilst $stat_{LC}(Cleanup)$ together with $stat_{LC}(Post_\tau)$ establishes Inv_τ. Thus correctness follows. Hippocraticness holds because target model pairs (or clusters) $t1, t2, \ldots$ are only deleted if they do not correspond to a source instance. Unique Instantiation can apply to multiple *exists* quantifiers in the same manner as to single quantifiers. Local hippocraticness applies to a source entity type instance and its corresponding target cluster. The principle of least change follows from the minimality of $stat_{LC}(C_i)$ as an activity to establish C_i. The treatment of Flattening/Unflattening is similar. The general form of the forward rule is

$$\begin{array}{l} E :: \\ \quad f : fs \ \& \ FCond(f) \implies \\ \qquad G \rightarrow exists(g| \\ \qquad\quad g.gId = f.fId \ \& \ g.eId = eId \ \& \\ \qquad\quad GCond(g) \ \& \ P(self, f, g)) \end{array}$$

The corresponding invariant is

$$\begin{array}{l} G :: \\ \quad GCond(self) \implies \\ \qquad E \rightarrow exists(e|e.eId = eId \ \& \\ \qquad\quad F \rightarrow exists(f|f.fId = gId \ \& \ f : e.fs \ \& \\ \qquad\qquad FCond(f) \ \& \ P(e, f, self))) \end{array}$$

$F \rightarrow select(FCond)$ is isomorphic to $G \rightarrow select(GCond)$, provided that every F instance occurs in some $e.fs$ collection for some $e : E$.

The simplified cleanup constraint is:

$$\begin{array}{l} G :: \\ \quad GCond(self) \ \& \\ \quad not(E \rightarrow exists(e|e.eId = eId \ \& \\ \qquad F \rightarrow exists(f|f.fId = gId \ \& \ FCond(f)))) \implies \\ \qquad\qquad\qquad self \rightarrow isDeleted() \end{array}$$

That is, $g : G$ is deleted if either its corresponding source element or source container are deleted. The case of a movement of an element from one container to another is handled by the $Post_\tau$ constraints, which update $G :: eId$ to re-establish Inv_τ. Other changes to the source model are also propagated to the target by $Post_\tau$, thus correctness holds. Hippocraticness holds since neither the cleanup or $Post_\tau$ constraints modify the target model if $Post_\tau$ & Inv_τ already holds.

For Inverse Recursive Descent (Figure 6), the invariant is:

$$\begin{array}{l} F :: \\ \quad FCond \implies E \rightarrow exists(e|e.eId = fId \ \& \\ \qquad ECond(e) \ \& \ e.esub = ESub[fsub.fId] \ \& \\ \qquad P(e, self)) \end{array}$$

From this the cleanup constraints can be derived as for Phased Construction above. The cleanup constraints propagate deletion of E instances to F instances. The operation calls propagate feature changes and object creation from the source to the target model – for elements that the operation is applied to. The specifier should ensure that the operation is always invoked on all instances of E satisfying $ECond$.

For the Map Objects before Links pattern (Figure 2), the construction of target instances is separated from the linking of these instances. In the case that S_i corresponds to T_j and $SRef$ to $TRef$ via one of the other patterns considered above, the linking constraints have the form:

$$\begin{array}{l} S_i :: \\ \quad T_j[idS_i].rr = TRef[r.idSRef] \end{array}$$

These define target model association ends rr from source model association ends r, looking-up target model elements $T_j[idS_i]$ and $TRef[r.idSRef]$ which have already been created by preceding constraint(s). The linking constraints can be inverted to a dual form which defines source data from target data as:

$$\begin{array}{l} T_j :: \\ \quad S_i[idT_j].r = SRef[rr.idTRef] \end{array}$$

according to Table 6.

The invariant for the linking constraint is:

$$\begin{array}{l} S_i :: \\ \quad T_j[idS_i].rr \subseteq TRef[r.idSRef] \end{array}$$

That is, every element of $t.rr$ should correspond to an element of $s.r$, if s and t correspond.

Cleanup constraints are not needed for linking constraints, because deletion of $SRef$ instances is propagated to deletion of $TRef$ instances by the cleanup constraint for $TRef$ construction (destruction of an instance includes removing it from every association end in which it resides, in UML-RSDS semantics). Other modifications to r are propagated to rr by the linking constraint itself.

By a similar argument to the previous cases, bx correctness, hippocraticness and least change can be shown. The re-establishment of $Post_\tau$ and Inv_τ for corresponding source and target elements is now performed by combination of the construction and linking $Post_\tau$ constraints.

REFERENCES

[1] A. Anjorin, T. Buchmann, and B. Westfechtel, "The families to persons case," in *Proc. 10th Transf. Tool Contest (TTC)*, A. Garcia-Dominguez, G. Hinkel, and F. Krikava, Eds., 2017.

[2] A. Anjorin, G. Varró, and A. Schürr, "Complex attribute manipulation in TGGs with constraint-based programming techniques," in *Proc. BX Electron. Commun. (EASST)*, vol. 49, 2012.

[3] A. Anjorin and M. Lauder, "A solution to the flowgraphs case study using triple graph grammars and eMoflon," *Electron. Proc. Theor. Comput. Sci.*, vol. 135, 2013, doi: 10.4204/EPTCS.135.8.

[4] F. Bancilhon and N. Spyratos, "Update semantics of relational views," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 557–575, 1981.

[5] M. Beine, N. Hames, J. Weber, and A. Cleve, "Bidirectional transformations in database evolution: A case study 'at scale,'" in *Proc. EDBT/ICDT*, 2014, pp. 100–107.

[6] G. Bergmann, C. Debreceni, I. Rath, and D. Varro, "Query-based access control for secure collaborative modeling using bidirectional transformations," in *Proc. MODELS*, 2016, pp. 351–361.

[7] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo, "Static fault localization in model transformations," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 490–506, May 2015.

[8] J. Bradfield and P. Stevens, "Enforcing QVT-R with mu-calculus and games," in *Fundamental Approaches to Software Engineering—FASE* (Lecture Notes in Computer Science), vol. 7793, V. Cortellessa and D. Varró, Eds. Berlin, Germany: Springer, 2013.

[9] J. Cabot, R. Clariso, E. Guerra, and J. de Lara, "Verification and validation of declarative model-to-model transformations through invariants," *J. Syst. Softw.*, vol. 8, no. 2, pp. 283–302, 2009.

[10] J. Cheney, J. McKinna, P. Stevens, and J. Gibbons, "Towards a repository of bx examples," in *Proc. EDBT/ICDT*, 2014, pp. 87–91.

[11] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "JTL: A bidirectional and change propagating transformation language," in *Software Language Engineering* (Lecture Notes in Computer Science), vol. 6563. 2011, pp. 183–202.

[12] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *Proc. GRACE Workshop*, 2008, pp. 260–283.

[13] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Constraint-driven modeling through transformation," *Softw., Syst. Model.*, vol. 14, no. 2, pp. 573–596, 2015.

[14] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer, "Information preserving bidirectional model transformations," in *Fundamental Approaches to Software Engineering*. 2007, pp. 72–86.

[15] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bi-directional tree transformations," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, p. 17, 2007.

[16] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner, "Towards verified model transformations," in *Proc. MODEVA*, 2006, pp. 78–93.

[17] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronisation," *Softw., Syst. Model.*, vol. 8, no. 1, pp. 21–43, 2009.

[18] H. Giese, S. Hildebrant, and S. Neumann, "Model synchronization at work: Keeping SysML and AUTOSAR models consistent," in *Proc. ICMT*, 2010, pp. 555–579.

[19] M. Gogolla and A. Vallecillo, "*Tractable* model transformation testing," in *Proc. 7th Eur. Conf. Modeling Found. Appl.*, 2011, pp. 221–235.

[20] T. Goldschmidt and G. Wachsmuth, "Refinement transformation support for QVT relational transformations," in *Proc. ENCS*, 2011, pp. 1–14.

[21] J. Greenyer and E. Kindler, *Softw. Syst. Model.*, vol. 9, p. 21, 2010.

[22] E. Guerra, J. de Lara, and F. Orejas, "Inter-modelling with patterns," *Softw., Syst. Model.*, vol. 12, no. 1, pp. 145–174, 2013.

[23] E. Guerra, J. de Lara, D. Kolovos, R. Paige, and O. M. dos Santos, "Engineering model transformations with transML," *Softw., Syst. Model.*, vol. 12, no. 3, pp. 555–577, 2013.

[24] E. Guerra and J. de Lara, "Colouring: Execution, debug and analysis of QVT-relations transformations through coloured Petri nets," *Softw., Syst. Model.*, vol. 13, no. 4, pp. 1447–1472, 2014.

[25] F. Hermann, N. Nachtigall, B. Braatz, S. Gottmann, and T. Engel, "Solving the FIXML2 code-case study with HenshinTGG," TTC, Toronto, ON, Canada, Tech. Rep., 2014.

[26] F. Hermann *et al.*, "Triple graph grammars in the large for translating satellite procedures," in *Proc. ICMT*, 2014, pp. 122–137.

[27] F. Hermann *et al.*, "Model synchronization based on triple graph grammars: Correctness, completeness and invertibility," *Softw., Syst. Model.*, vol. 14, no. 1, pp. 241–269, 2015.

[28] T. Hettel, M. Lawley, and K. Raymond, "Model synchronisation: Definitions for round-trip engineering," in *Proc. ICMT*, 2008, pp. 31–45.

[29] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu, "Feature-based classification of bidirectional transformation approaches," *Softw., Syst. Model.*, vol. 15, no. 3, pp. 907–928, 2016.

[30] Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser, "Code generation by model transformation: A case study in transformation modularity," *Softw., Syst. Model.*, vol. 9, no. 3, pp. 375–402, 2010.

[31] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, nos. 1–2, pp. 31–39, 2008.

[32] M. Kleiner, M. D. Del Fabro, and D. De Santos, "Transformation as search," in *Proc. ECMFA*, in Lecture Notes in Computer Science, vol. 7949. 2013, pp. 54–69.

[33] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The Epsilon transformation language," in *Proc. ICMT*, 2008, pp. 46–60.

[34] M. Kramer and K. Rakhman, "Automated inversion of attribute mappings in BX," in *Proc. 5th Int. Workshop Bidirectional Transformations (Bx)*, A. Anjorin and J. Gibbons, Eds. ETAPS, 2016.

[35] K. Lano and S. Kolahdouz-Rahimi, "Constraint-based specification of model transformations," *J. Syst. Softw.*, vol. 88, no. 2, pp. 412–436, Feb. 2013.

[36] K. Lano. (2018). *The UML-RSDS Manual*. [Online]. Available: www.dcs.kcl.ac.uk/staff/kcl/uml2web/umlrsds.pdf

[37] K. Lano, T. Clark, and S. Kolahdouz-Rahimi, "A framework for model transformation verification," *Formal Aspects Comput.*, vol. 27, no. 1, pp. 193–235, 2015.

[38] K. Lano and S. Kolahdouz-Rahimi, "Model-transformation design patterns," *IEEE Trans. Softw. Eng.*, vol. 40, no. 12, pp. 1224–1259, Dec. 2014.

[39] K. Lano, S. Kolahdouz-Rahimi, and S. Yassipour-Tehrani, "Patterns for specifying bidirectional transformations in UML-RSDS," in *Proc. ICSEA*, 2015.

[40] K. Lano, S. Kolahdouz-Rahimi, and S. Yassipour-Tehrani, "Model transformation semantic analysis by transformation," in *Proc. VOLT*, 2015.

[41] K. Lano and S. Yassipour-Tehrani, "Verified bidirectional transformations by construction," in *Proc. VOLT*, 2016, pp. 28–37.

[42] K. Lano, *Agile Model-Based Development Using UML-RSDS*. Boca Raton, FL, USA: CRC Press, 2016.

[43] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, and M. Sharbaf, "A survey of model transformation design pattern usage," in *Proc. ICMT*, 2017, pp. 108–118.

[44] K. Lano, S. Yassipour-Tehrani, H. Alfraihi, and S. Kolahdouz-Rahimi, "Translating UML-RSDS OCL to ANSI C," in *Proc. OCL*, 2017, pp. 317–330.

[45] K. Lano, S. Kolahdouz-Rahimi, M. Sharbaf, and H. Alfraihi, "Technical debt in model transformation specifications," in *Proc. ICMT*, 2018, pp. 127–141.

[46] N. Macedo and A. Cunha, "Least-change bidirectional model transformation with QVT-R and ATL," *Softw., Syst. Model.*, vol. 15, no. 3, pp. 783–810, 2014, doi: 10.1007/s10270-014-0437-x.

[47] K. Matsuda *et al.*, "Bidirectionalization transformation based on automatic derivation of view complement functions," in *Proc. ICFP*, 2007, pp. 47–58.

[48] L. Meertens, "Designing constraint maintainers for user interaction," in *Proc. 3rd Workshop Program. Structured Documents, Tokyo Univ.*, 2005.

[49] C. Mokaddem, H. Sahraoui, and E. Syriani, "Towards rule-based detection of design patterns in model transformations," in *System Analysis and Modeling. Technology-Specific Aspects of Models* (Lecture Notes in Computer Science), vol. 9959. 2016, pp. 211–225.

[50] C. Morgan and K. Robinson, "Specification statements and refinement," *IBM J. Res. Develop.*, vol. 31, no. 5, pp. 546–555, Sep. 1987.

[51] Object Management Group. (2016). *MOF 2.0 Query/View/Transformation Specification V1.3*. [Online]. Available: https://www.omg.org/spec/QVT/About-QVT/

[52] *Object Constraint Language Specification V2.4*, OMG, 2014.

[53] *Action Language for Foundational UML (ALF), V1.0.1*, OMG, 2015.

[54] L. Paolini, M. Piccolo, and L. Roversi, "A class of reversible primitive recursive functions," *Electron. Notes Theor. Comput. Sci.*, vol. 322, pp. 227–242, Apr. 2016.

[55] S. Peldszus *et al.*, "Java refactoring case using eMoflon," TTC, Toronto, ON, Canada, Tech. Rep., 2015.

[56] *RTCA/EUROCAE DO-178C Standard: Software Considerations in Airborne Systems and Equipment Certification*, RTCA, 2012.

[57] L. Samimi-Dehkordi, B. Zamani, and S. Kolahdouz-Rahimi, "EVL+Strace: A novel bidirectional model transformation approach," *Inf. Softw. Technol.*, vol. 100, pp. 47–72, Aug. 2018.

[58] O. Semerath, C. Debreceni, Á. Horváth, and D. Varró, "Incremental backward change propagation of view models by logic solvers," in *Proc. MODELS*, 2016, pp. 306–316.

[59] P. Stevens, "Bidirectional model transformations in QVT: Semantic issues and open questions," *Softw., Syst. Model.*, vol. 9, no. 1, pp. 7–20, Jan. 2010.

[60] P. Stevens, "A simple game-theoretic approach to checkonly QVT-relations," *Softw., Syst. Model.*, vol. 12, no. 1, pp. 175–199, 2013.

[61] J. Voigtlander, Z. Hu, K. Matsuda, and M. Wang, "Combining syntactic and semantic bidirectionalization," in *Proc. ICFP*, 2010, pp. 181–192.

[62] M. Wang, J. Gibbons, and N. Wu, "Incremental updates for efficient bidirectional transformations," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 392–403, 2011.

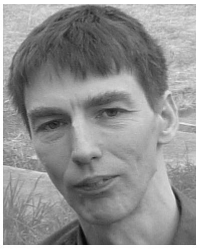
[63] E. D. Willink, "The micromapping model of computation," in *Proc. ICMT*, 2017.

[64] E. Willink. (2018). *The QVT-D Project*. [Online]. Available: https://projects.eclipse.org/projects/modeling.mmt.qvtd

[65] Y. Xiong *et al.*, "Towards automatic model synchronization from model transformationss," in *Proc. ASE*, 2007, pp. 164–173.

[66] Y. Xiong, H. Song, Z. Hu, and M. Takeichi, "Synchronizing concurrent model updates based on bidirectional transformation," *Softw., Syst. Model.*, vol. 12, no. 1, pp. 89–104, 2013.

[67] A. Yie, R. Casallas, D. Deridder, and D. Wagelaar, "Realizing model transformation chain interoperability," *Softw., Syst. Model.*, vol. 11, no. 1, pp. 55–75, 2012.



KEVIN 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 modeling, and in applying software engineering principles to transformation construction. In recent years, he has worked on the integration of model-based development and agile development.

SHEKOUFEH KOLAHDOUZ-RAHIMI received the Ph.D. degree in computer science from Kings College London, in 2013. She is an Assistant Professor with the Computer Engineering Department, University of Isfahan, where she is an Active Member of the Model Driven Software Engineering Research Group. Her research interests include design patterns for model transformation, specification and verification of model transformations, bidirectional model transformations, domain-specific modeling languages, and search-based software engineering.

SOBHAN YASSIPOUR-TEHRANI has been working on requirements engineering (RE) in model transformations (MT). He has developed a RE technique suitability framework for MT projects by which the developer(s) are able to select the most appropriate RE technique for a particular requirement or a set of requirements. His research interests include RE, MT, goal modeling, and agile software development.