Incremental Bidirectional Transformations: Applying QVT Relations to the Families to Persons Benchmark

Bernhard Westfechtel
Applied Computer Science I, University of Bayreuth, D-95440 Bayreuth, Germany

Keywords: Model Transformation, Bidirectional Transformation, Benchmark, QVT Relations.

Abstract: Model transformations constitute a key technology for model-driven software engineering. In round-trip engineering processes, model transformations are performed not only in forward, but also in backward direction. Since defining forward and backward transformations separately is both awkward and error-prone, bidirectional transformation languages provide a single definition for both directions. This paper evaluates the transformation language QVT Relations (QVT-R) which allows to specify incremental bidirectional transformations — as required for round-trip engineering for propagating changes in both directions — declaratively at a high level of abstraction. We apply QVT-R to a well-known benchmark example, the Families to Persons case. This case study demonstrates a number of limitations of the QVT-R language which result from the strictly state-based design of the language as well as from the way in which the semantics of QVT-R transformations are defined.

1 INTRODUCTION

Model transformations (Czarnecki and Helsen, 2006) constitute a key technology for model-driven software engineering. A model transformation takes a set of source models as input and creates or updates a set of target models. A transformation operates in *batch* mode if it generates the target models from scratch; in contrast, an *incremental* transformation propagates changes from sources to targets. Furthermore, we may distinguish between *unidirectional* transformations which are executed only from source to target, and *bidirectional transformations* which are executed also in the opposite direction. Bidirectional transformations occur in a wide variety of application domains (Czarnecki et al., 2009), including, but not restricted to model-driven software engineering. In this paper, we focus specifically on incremental bidirectional transformations, as they are required e.g. in round-trip engineering processes.

A *bidirectional transformation language* is a language which allows to define bidirectional transformations with the help of a single transformation definition being executable in both directions; see (Hidaka et al., 2016) for a survey. A bidirectional transformation language promises to save specification effort since the transformation developer does not need to code each direction separately. Furthermore, a bidirectional transformation language may assist in defining mutually consistent forward and backward transformations. For example, the forward and backward transformations may satisfy certain round-trip laws such as that the execution of a backward transformation, immediately following a forward transformation, leaves the original source model untouched (Foster et al., 2007).

QVT Relations (*QVT-R*) is a model transformation language which was defined by the Object Management Group (OMG) as part of the QVT standard (Object Management Group, 2016a). QVT-R is based on the Meta Object Facility (MOF (Object Management Group, 2016b)) for defining models as instances of metamodels, and the Object Constraint Language (OCL (Object Management Group, 2014)) for expressing queries and constraints on MOF models. QVT-R is a declarative language which allows to specify consistency relationships between patterns. In QVT-R, a transformation developer may define a bidirectional transformation which may be executed in different ways: in forward or backward direction, in batch or incremental mode, as well as in checkonly or enforcing mode. In checkonly mode, the consistency between a source and a target model is checked; in enforcing mode, the target model is updated (or created anew) to make it consistent with the source model. Thus, a single declarative transformation defini-

Westfechtel, B.
Incremental Bidirectional Transformations: Applying QVT Relations to the Families to Persons Benchmark.
DOI: 10.5220/0006679700390053
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 39-53
ISBN: 978-989-758-300-1

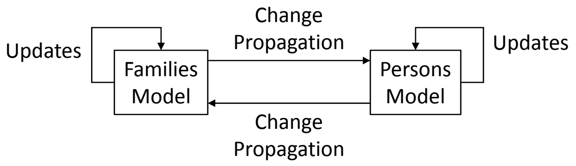

Figure 1: The Families to Persons case.

tion may be executed in up to six different ways. This approach promises to make the development of bidirectional transformations considerably easier.

In the light of the diversity of bidirectional transformation languages, the need for benchmarks has been identified early (Czarnecki et al., 2009). Later, a proposal for structuring benchmarks for bidirectional transformations was published (Anjorin et al., 2014). Only recently, this proposal was materialized into an implementation called Benchmarx, which constitutes a practical benchmark framework for bidirectional transformations (Anjorin et al., 2017b). Based on this framework, a popular case from the literature — the Families to Persons case — was implemented (Anjorin et al., 2017a). In this paper, we evaluate incremental bidirectional transformations in QVT-R with the help of the Families to Persons benchmark case as described in (Anjorin et al., 2017a).

The Families to Persons benchmark case is illustrated in Figure 1. In this informal diagram, boxes represent models, while arrows stand for model updates. Two related, but differently structured models have to be kept consistent: A families model with parents and children, and a persons model containing a flat set of males and females. Updates may be performed on both models, and have to be propagated in both directions. While the Families to Persons case is rather small and thus implementable with acceptable effort, it poses a number of challenges such as heterogeneous metamodels, loss of information, the absence of keys (uniquely identifying properties of model elements), non-determinism, configurability (of the backward transformation), renamings and moves, order dependent update behavior, and application-specific requirements to change operations.

Our contribution consists in the solution of the Families to Persons case in QVT-R and its evaluation. Surprisingly, our evaluation reveals a number of limitations, resulting in a large number of failed test cases. A detailed examination shows that these failures are due the semantic rules determining the execution behavior of QVT-R transformation definitions. The problems are not primarily caused by ambiguities or contradictions in the semantics definition. Rather, the Persons to Families case demands for behavior that deviates from the semantics which QVT-R incorporates. Thus, our main intent is to clearly elaborate the reasons for failures in the benchmark and discuss the lessons learned from this case study.

In particular, we identified the following problems when applying QVT-R to the Families to Persons case:

Imprecise Change Propagation. Due to the strictly state-based design and the absence of persistent traces, changes may be propagated only imprecisely. For example, if the name of a family member is changed, the corresponding person is deleted and recreated, implying that the birthday is lost.

Unidirectional Transformations. Although QVT-R supports the specification of bidirectionally executable transformations, our best-effort solution requires to write two unidirectional transformations because the rules for forward and backward transformations differ significantly.

Non-injective Mappings. Due to the check-before-enforce semantics of QVT-R, multiple family members sharing the same name would be mapped onto the same person (and vice versa) unless the transformation definition is written in such a way that an injective mapping is enforced. Unfortunately, enforcing injective mappings turns out to be awkward, and it requires pairs of unidirectional transformations (see above).

Duplicate Transformations. QVT-R applies all rules to all matches, without checking for conflicts. This may result in undesirable behavior. For example, in the backward transformation from the persons model to the families model there are two rules for mapping females to mothers and daughters, respectively. The transformation developer must ensure that the preconditions for applying these rules are mutually exclusive.

The rest of this paper is structured as follows: Section 2 introduces QVT-R. Section 3 describes the Families to Persons case, which is solved in Section 4. Section 5 evaluates the solution. Section 6 discusses related work, and Section 7 concludes the paper.

2 QVT RELATIONS

This section summarizes some global properties of QVT Relations (QVT-R); the language constructs will be explained on demand in parallel to the solutions presented in Section 4. For a detailed description of QVT-R, the reader is referred to the standard (Object Management Group, 2016a).

QVT-R is a declarative language for specifying transformations between models which are typed by

Table 1: Properties of QVT-R.

Property	Explanation
State-based	Transformation depends only on model states, not on deltas
Trace-less	Transformation does not rely on a persistent trace
$n:1$ *mappings*	Multiple source pattern instances may be mapped to the same target pattern instance
Non-conflicting relations	Relations are applied independently of each other

MOF-based metamodels. In general, any number n of models may be involved in a transformation; in the following explanations, we will assume $n = 2$. A *transformation* is defined in terms of *relations* between patterns in the participating models.

A transformation is executed in the direction of a specific model. A transformation is *unidirectional* if it can be executed only in a specific direction; it is *bidirectional* if it can be executed in both directions. In the latter case, the transformation definition must satisfy a number of restrictions (dataflow constraints) ensuring that the transformation is actually executable in both directions. QVT-R does not distinguish explicitly between uni- and bidirectional transformations.

QVT-R distinguishes between two modes of transformation executions. In *checkonly mode*, it is checked whether the target model is consistent with the source model. In *enforcing mode*, the target model is updated in order to make it consistent with the source model. With respect to enforcing transformations, *batch transformations* are considered as a special case of *incremental transformations* (initially, the target model is empty).

QVT-R transformations are based on a *for-all-exists semantics*: In checkonly mode, it is checked for each relation and each match of a source pattern whether a match of the corresponding target pattern exists in the target model. In enforcing mode, a match of the target pattern is created when it does not exist yet. The for-all-exists semantics results in $n:1$ *mappings*, i.e., it is allowed that multiple matches in the source model are mapped to the same match in the target model. This statement holds both within a specific relation and across different relations.

Due to the for-all-exists semantics, a directed check reports consistency of the target model with the source model even if the target model contains “too many” elements. Therefore, checking mutual consistency requires two directed checks. In contrast, in enforcing mode all unmatched target elements are deleted to ensure mutual consistency.

In QVT-R, all relations are *non-conflicting*. Thus, if two relations are applicable to the same match, both of them are applied. If it is intended that at most one of them is executed, the transformation developer must specify the application conditions accordingly. Similarly, matches may have arbitrary overlaps — both within a single relation and across multiple relations.

QVT-R follows a purely *state-based approach* to incremental transformations: The only information which is assumed to be present are the states of the source and the target model. In this way, the prerequisites for the execution of incremental transformations are minimized. Thus, for any pair of models s and t, an incremental transformation for updating t may be executed, regardless of the history of changes.

In particular, QVT-R does not assume the presence of *deltas*, which record sequences of change operations on the source model and could be exploited to improve change propagation. Nor does QVT-R rely on *(persistent) traces*, which record the relationships between source and target model elements of previous transformation executions. Thus, in QVT-R an incremental transformation from s to t may be performed even if s and t have been created independently and no knowledge regarding the changes on s is available.

Finally, incremental QVT-R transformations rely on a *check-before-enforce semantics*: If a match of a source pattern of some relation is located, it is checked first whether a match of the corresponding target pattern already exists. In this case, the match of the source pattern is related to the already existing match of the target pattern. The check-before-enforce semantics differs from a trace-based semantics, where a new target pattern instance is created when the respective source pattern instance has not been transformed yet.

Table 1 summarizes essential properties of QVT-R being relevant to the case studied in this paper.

3 FAMILIES TO PERSONS

Different variants of the Families to Persons case have been proposed. To make this paper self-contained, we describe the variant on which our work is based, following (Anjorin et al., 2017a). The case is rather small such that it is implementable with acceptable effort. On the other hand, it includes several challenges to be summarized at the end of this section.

Below, the Families to Persons case is described in an informal way (apart from the definition of metamodels, see below). While informal descriptions suffer from a lack of precision, we refrain from a formaliza-

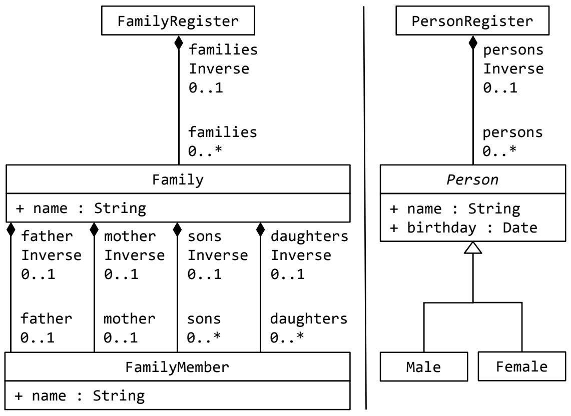

Figure 2: Metamodels.

tion, e.g., with the help of finite model theory (Libkin, 2004), which constitutes a challenge on its own and considerably goes beyond the scope of this paper.

3.1 Metamodels and Consistency

The Families to Persons case deals with the synchronization of a families model with a persons model. The underlying *metamodels* are displayed in Figure 2 as class diagrams. For metamodeling, we employ Ecore - an implementation of Essential MOF, a subset of MOF, provided by the Eclipse Modeling Framework (Steinberg et al., 2009). Please note that we adopt the following terminology: A *model* represents a collection of instance data. A *metamodel* is a "model for models" and defines the types of objects, attributes, and links, as well as the rules for their composition.

We assume a unique root in each model. A family register stores a collection of families. Each family has members who are distinguished by their roles. The metamodel allows for at most one father and at most one mother as well as an arbitrary number of daughters and sons. A person register maintains a flat collections of persons who are either male or female. Note that key properties may be assumed in neither model: There may be multiple families with the same name, name clashes are even allowed within a single family, and there may be multiple persons with the same name and birthday.

A families model is *consistent* with a persons model if a bijective mapping between family members and persons can be established such that (i) mothers and daughters (fathers and sons) are paired with females (males), and (ii) the name of every person p is "$f.name$, $m.name$", where m is the member (in family f) paired with p.

3.2 Batch Transformations

After running a transformation in any direction, it is required that the participating models are mutually consistent according the definition given above. However, this requirement does not suffice to define the functionality of transformations in a unique way. Below, we first consider *batch transformations*, where the target model is created from scratch.

The functionality of a *forward transformation* is straightforward: Map each family member to a person of the same gender and compose the person's name from the family name and the first name; the birthday remains unset. The backward transformation is more involved: A person may be mapped either to a parent or a child, and persons may be grouped into families in different ways.

To reduce non-determinism, two Boolean parameters control the backward transformation, resulting in a *configurable backward transformation*: `preferParentToChild` defines whether a person is mapped to a parent or a child. `preferExistingToNewFamily` determines whether a person is added to an existing family (if available), or a new family is created along with a single family member. If both parameters are set to true, the second parameter takes precedence: If a family is available with a matching family name, but there is no matching family with an unoccupied parent role, the member is inserted into an existing family as a child.

3.3 Incremental Transformations

For *incremental transformations*, updates such as insertions, deletions, changes of attribute values, and move operations have to be considered. In *forward direction*, insertion of a family has no effect on the target model. Insertion of a member results in insertion of a corresponding person; likewise for deletions. If a family is deleted, all persons corresponding to its members are deleted. If a member is renamed, the corresponding person is renamed accordingly. If a family is renamed, all persons corresponding to family members are renamed. If a member is moved, different cases have to be distinguished. If the gender is retained, the corresponding person object is preserved; otherwise, it is deleted, and a new person object with a different gender is created whose attributes are copied from the old person object. A local move within a family does not affect the corresponding person's name; a move to another family results in a potential update of the person's name.

In *backward direction*, the effect of an update depends on the values of the configuration parameters which may vary between different transformation executions. Please note that the parameter settings must not affect already established correspondences; rather, they apply only to future updates. Deletion of a person propagates to the corresponding family member. If a person is inserted, it depends on the configuration parameters how insertion propagates to the families model (see above). Persons cannot be moved because the persons model consists of a single, flat, and unordered collection. Changes of birthdays do not propagate to the opposite model. If the first name of a person is changed, the first name of the corresponding family member is updated accordingly. Finally, if the family name of a person is changed, this change does not affect the current family and its members: The family preserves its name even if it does not contain other members; thus, the update has no side effects on the existing family. Rather, the corresponding family member is moved to another family, which may have to be created before the move; the precise update behavior depends on the parameter settings.

3.4 Transformation Laws

The transformations developed for the Families 2 Persons benchmark should satisfy a set of general *laws* (Stevens, 2010). First, the result of a transformation from s to t should be *correct*, i.e., s and t should be mutually consistent (see Section 3.1). Furthermore, transformations should be *hippocratic*: The target model should not be changed if consistency has already been established. Thus, an immediate re-run of a transformation establishing correctness should have no effect (even if configuration paramaters are switched in the backward transformation). Finally, *round-trip laws* (Foster et al., 2007) should be satisfied: A backward transformation following a forward transformation should leave the original source model untouched; likewise for a forward transformation following a backward transformation.

3.5 Challenges

The Families to Persons case includes a number of *challenges* which are summarized below:

Heterogeneous Metamodels. The transformation has to perform a mapping between heterogeneous metamodels, where the same information is represented in different ways (concerning e.g. names and genders).

Loss of Information. The family structure is lost in the forward transformation; birthdays are lost in the backward transformation.

No Keys. There are no uniquely identifying properties for family members or persons, which makes propagation of changes difficult.

Non-determinism. The families model does not contain information for defining birthdays in the persons model. Conversely, the persons model does not contain information about the family structure. The forward and backward transformations have to resolve non-determinism.

Configurability. Non-determinism in the backward transformation is handled by configuration parameters which affect the family structure. The parameters may be changed from execution to execution and should affect only new elements of the person model in each run.

Renamings and Moves. Change operations to be propagated include not only creations and deletions, but also renamings and moves which must not be reduced to deletions and creations (otherwise, the principle of *least change* (Macedo and Cunha, 2016), which demands that the changes are propagated in a minimally invasive way, would be violated).

Order-dependent Update Behavior. The backward transformation depends on the order in which changes to the persons model are processed. For example, if two persons are to be inserted into the same family as parents, the first one will be inserted as parent, and the second one will follow as a child.

Specific Requirements to Change Operations. The case description includes specific requirements to change operations. For example, if the family name of a person is changed, the person should be moved to another family (rather than having the family name updated, with side effects on other members).

4 SOLUTION

In the following, we will first present an initial solution which is easy to understand (Section 4.1). Since this proposal suffers from several deficits, we will refine it into an improved, more sophisticated solution. In passing, we will mention a number of issues which we faced in developing the solutions; these will be summarized and discussed systematically in Section 5.

Listing 1: QVT-R transformation, initial version.

```
1  transformation families2persons_bx (famDB : Families, perDB : Persons) {
2
3      top relation FamilyRegister2PersonRegister {
4          enforce domain famDB familyRegister : Families::FamilyRegister {};
5          enforce domain perDB personRegister : Persons::PersonRegister {};
6          where {
7              Father2Male(familyRegister, personRegister);
8              Son2Male(familyRegister, personRegister);
9              Mother2Female(familyRegister, personRegister);
10             Daughter2Female(familyRegister, personRegister);
11         }
12     }
13
14     relation Father2Male {
15         familyName, firstName, fullName : String;
16         enforce domain famDB familyRegister : Families::FamilyRegister {
17             families = family : Families::Family {
18                 name = familyName,
19                 father = father : Families::FamilyMember {
20                     name = firstName
21                 }
22             }
23         };
24         enforce domain perDB personRegister : Persons::PersonRegister {
25             persons = male : Persons::Male {
26                 name = fullName
27             }
28         };
29         where {
30             fullName = familyName + ', ' + firstName;
31             firstName = firstName(fullName);
32             familyName = familyName(fullName);
33         }
34     }
35     ...
36 }
```

4.1 Initial Version

Listing 1 shows a straightforward *bidirectional transformation*. The *candidate models* — models involved in the transformation — are declared in line 1. Please notice that there is no distinction between inputs and outputs; e.g., the families model serves as input for the forward transformation and as output for the backward transformation.

The transformation definition consists of five *relations*, two of which are shown in Listing 1. The relation starting in line 3 is a *top-level relation*, i.e., it is not called in any relation and is applied automatically to each match located in the source model. This relation serves to relate the root elements of the families and persons models. The source and target patterns of a relation are defined by *domains*. Each domain belongs to one of the candidate models (e.g., the families model `famDB` in line 4). Furthermore, it has a unique root node which is typed (`familyRegister`). The modifier `enforce` allows to use the domain for enforcement (instantiation of the domain in the target model). Since both domains in lines 4–5 are marked by this modifier, the relation may be enforced in both transformation directions. Finally, further relations are called in the `where` clause (lines 7–10). Only one of these relations (`Father2Male`) is shown; the other relations are defined analogously.

The relation starting in line 14 is a *called relation*, which is applied only when it is called explicitly. Then, it is applied to all matches of its source pattern for the root element fixed in the call. Thus, `Father2Male` is applied to all fathers in the families register (*implicit iteration*). Each father is mapped to a male person whose name is composed from the family name and the first name. If executed in backward direction, the person is mapped to a father; the name is decomposed accordingly. The variables declared in line 15 are used for storing the values of the name attributes involved in the transformation. The equations in the `where` clause (lines 30–32) compose or decompose name attributes, depending on the transformation direction. In forward direction, the equation in line 30 binds the variable `fullName`; the following equations perform redundant checks. In backward direction, `firstName` and `lastName` are bound in lines 31–32, followed by a redundant check in line 30. Please notice that the order of execution of equations is determined by dataflow constraints, not by the textual order (which is immaterial). All equations are executed in each direction, but perform different tasks (binding of a variable vs. checking of a constraint).

The transformation definition presented in Listing 1 is executable in both directions and easy to understand (compared to the transformation definitions to be presented below). However, it suffers from two fundamental flaws which will be addressed by the improved version presented below. First, due to QVT-R's check-before-enforce semantics, mappings are $n : 1$ rather than $1 : 1$: For example, different members in the families model sharing the same family name and first name will be mapped to the same person in the persons model. Second, in backward direction each person will be transformed twice, both as a parent and as a child.

4.2 Improved Version

When implementing a bidirectional transformation in QVT-R, the transformation developer may either provide a single transformation definition which is executable in both directions (see above), or may create two unidirectional transformation definitions. For full consideration of the requirements in the Families to Persons case, we have to resort to the latter option because the rules for forward and backward transformations differ significantly. These differences result from the fact that the backward transformation needs to be configurable with respect to the mapping of persons to parents or children and to existing or new families, respectively. Furthermore, the enforcement of $1 : 1$ mappings is realized with the help of rules which can be executed only in one direction.

4.2.1 Forward Transformation

Listing 2 shows a unidirectional forward transformation which enforces $1 : 1$ mappings. In contrast to the bidirectional transformation definition in Listing 2, domains in the families model are marked as `checkonly`, indicating that they may not be used for enforcement. In the transformation definition, we make use of a language construct which has not been mentioned so far: *Primitive domains* (e.g., line 19) are parameters which are passed to called relations; here, we employ them for passing collections.

Enforcing injective mappings requires a procedural style of specification: The collection of elements which have been matched or created so far is supplied as parameter to called relations; a postcondition ensures that the `where` clause fails if an element is reused according to the check-before-enforce semantics. The failing postcondition enforces the creation or matching of a new element.

From the top-level relation, the relation `Member2-Person` is called. Please notice that members are processed sequentially; this is achieved by calling the relation on a specific member (lines 9, 49). Furthermore, the relation is supplied with two collections: the collection of other members which still need to be processed (line 19), and the collection of persons which have already been created or matched (line 21). These parameters are passed on to the relations handling the specific cases. Thus, the relation `Member2Person` acts as a switch; please notice that exactly one of the called relations will be applicable, such that sequential processing of matches is retained. From the specific relations, only `Daughter2Female` is shown. In its `where` clause, line 46 prevents reuse of a person object in spite of the check before enforce semantics: The person to be created or matched must not be contained in the collection of elements having been gathered so far. In the recursive call of `Member2Person`, the new element is added to the collection of persons which was passed as parameter. Similarly, the collection of members which still need to be processed is reduced by the current member.

While this approach works, it is a laborious way to guarantee injective mappings. Furthermore, in the opposite direction the roles of the collections passed as parameters need to be switched. As a consequence, different transformation definitions are needed for different directions.

4.2.2 Backward Transformation

To enforce injective mappings, we may apply the same method in the backward transformation as in the forward transformation. To simplify matters and to avoid redundancies in the presentation, we will provide a simplified backward transformation which does not enforce injective mappings, but avoids duplicate transformations of persons into parents and children (see remark at the end of Section 4.1). Furthermore, the backward transformation is *configurable* with respect to the mappings to parents or children and existing or new families, respectively. Excerpts from the transformation definition are displayed in Listings 3 and 4.

Since only models may be passed as parameters to transformations, we introduce a *configuration model*

Listing 2: Forward transformation.

```
transformation families2persons (famDB : Families, perDB : Persons) {
    top relation FamilyRegister2PersonRegister {
        members : Sequence(Families::FamilyMember);
        checkonly domain famDB familyRegister : Families::FamilyRegister {};
        enforce domain perDB personRegister : Persons::PersonRegister {};
        where {
            members = allMembers(familyRegister);
            if members->size() > 0 then
                Member2Person(members->first(),
                              members->excluding(members->first()),
                              personRegister, Sequence { })
            else
                true
            endif;
        }
    }
    relation Member2Person {
        checkonly domain famDB member : Families::FamilyMember {};
        primitive domain members : Sequence(Families::FamilyMember);
        enforce domain perDB personRegister : Persons::PersonRegister {};
        primitive domain persons : Sequence(Persons::Person);
        where {
            Father2Male(member, members, personRegister, persons);
            Son2Male(member, members, personRegister, persons);
            Mother2Female(member, members, personRegister, persons);
            Daughter2Female(member, members, personRegister, persons);
        }
    }
    ...
    relation Daughter2Female {
        firstName, familyName, fullName : String;
        checkonly domain famDB member : Families::FamilyMember {
            name = firstName,
            daughtersInverse = family : Families::Family {
                name = familyName
            }
        };
        primitive domain members : Sequence(Families::FamilyMember);
        enforce domain perDB personRegister : Persons::PersonRegister {
            persons = female : Persons::Female {
                name = fullName
            }
        };
        primitive domain persons : Sequence(Persons::Person);
        where {
            persons->excludes(female);
            fullName = familyName + ', ' + firstName;
            if members->size() > 0 then
                Member2Person(members->first(),
                              members->excluding(members->first()),
                              personRegister, persons->append(female))
            else
                true
            endif;

        }
    }
}
```

Listing 3: Backward transformation (part 1).

```
 1  transformation persons2families
 2                (perDB : Persons, conf : Config, famDB : Families) {
 3      top relation FamilyRegister2PersonRegister {
 4          preferParent, preferExisting : Boolean;
 5          checkonly domain perDB personRegister : Persons::PersonRegister {};
 6          checkonly domain conf configuration : Config::Configuration {
 7              preferParentToChild = preferParent,
 8              preferExistingToNewFamily = preferExisting
 9          };
10          enforce domain famDB familyRegister : Families::FamilyRegister {};
11          where {
12              Male2Member
13                  (personRegister, preferParent, preferExisting, familyRegister);
14              Female2Member
15                  (personRegister, preferParent, preferExisting, familyRegister);
16          }
17      }
18      relation Female2Member {
19          fullName, firstName, familyName : String;
20          mapToMother : Boolean;
21          checkonly domain perDB personRegister : Persons::PersonRegister {
22              persons = female : Persons::Female {
23                  name = fullName
24              }
25          };
26          primitive domain preferParent : Boolean;
27          primitive domain preferExisting : Boolean;
28          enforce domain famDB familyRegister : Families::FamilyRegister {};
29          where {
30              firstName = firstName(fullName);
31              familyName = familyName(fullName);
32              mapToMother = mapToMother(familyRegister,
33                                        preferParent, preferExisting,
34                                        firstName, familyName);
35              if mapToMother then
36                  Female2Mother(female,
37                                preferExisting, firstName,
38                                familyName, familyRegister)
39              else
40                  Female2Daughter(female,
41                                  preferExisting, firstName,
42                                  familyName, familyRegister)
43              endif;
44          }
45      }
46      ...
```

(parameter `conf` in line 2 in Listing 3) which contains a single configuration object storing the values of the configuration parameters. Accordingly, the top-level relation for mapping family to person registers is extended with a third domain from which the parameter values are read (lines 6–9). These values are passed on to called relations. Please recall that QVT-R allows to define transformations among more than two models; here, two models serve as input, and one model is created or updated as output.

From the top-level relation, two relations are called which are used to map males and females, respectively. These relations serve as switches for deciding whether a person should be mapped either to a parent or to a child. This decision is performed in the `where` clause. For example, in the relation `Female2Member` the predicate `mapToMother` is evaluated which depends on the current state of the family register, the configuration parameters, and the first name and the family name of the female to be transformed (lines 32–34). Depending on the result of the evaluation, the female is mapped to a mother or to a daughter by calling corresponding relations `Female2Mother` or `Female2Daughter`, respectively.

Listing 4: Backward transformation (part 2).

```
    ...
    query mapToMother(familyRegister : Families::FamilyRegister,
            preferParent : Boolean, preferExisting : Boolean,
            firstName : String, familyName : String) : Boolean {
        if matchingMotherExists(familyRegister, firstName, familyName) then
            preferParent or
            not matchingDaughterExists(familyRegister, firstName, familyName)
        else
            not matchingDaughterExists
                    (familyRegister, firstName, familyName) and
            preferParent and
            not (preferExisting and
                 matchingFamilyExists(familyRegister, familyName) and
                 not matchingFamilyWithoutMotherExists
                         (familyRegister, familyName))
        endif
    }
    relation Female2Mother {
        checkonly domain perDB female : Persons::Female {};
        primitive domain preferExisting : Boolean;
        primitive domain firstName : String;
        primitive domain familyName : String;
        enforce domain famDB familyRegister : Families::FamilyRegister {
            families = family : Families::Family {
                name = familyName,
                mother = mother : Families::FamilyMember {
                    name = firstName
                }
            }
        };
        when {
            mother =
                if matchingMotherExists
                        (familyRegister, firstName, familyName) then
                    matchingMother(familyRegister, firstName, familyName)
                else
                    mother
                endif;
            family =
                if not mother.oclIsUndefined() then
                    mother.motherInverse
                else
                    if preferExisting and
                        matchingFamilyWithoutMotherExists
                            (familyRegister, familyName)
                    then
                        matchingFamilyWithoutMother(familyRegister, familyName)
                    else
                        family
                    endif
                endif;
        }
    }
    ...
}
```

The predicate `mapToMother` is defined by an OCL query (lines 2–17 in Listing 4) which realizes the mapping rules explained in Section 3.2. As mentioned in Section 3.3, these rules should become effective only for new persons; the previous mappings should remain untouched. Thus, it is checked first whether a matching member already exists in the families model. For example, if the transformation is configured such that children are preferred and a matching mother is available (but not a matching daughter), the current female is mapped to a mother rather than to a daughter.

Finally, four relations are responsible for actually performing the required mapping. For example, `Female2Mother` (lines 18–53) maps a female to a mother. This relation is equipped with a `when` clause which is executed before the target domain is instantiated (lines 31–52). If the `when` clause were omitted, either the check-before-enforce semantics would apply, resulting in a reuse of both the family and the mother, or both elements would be instantiated anew. In this way, it would not be possible to insert a new mother into an existing family. Therefore, the `when` clause contains two equations which conditionally bind the variables `mother` and `family` before the target domain is processed. If the respective binding conditions do not hold, the variables retain their previous bindings, i.e., they remain unbound. If the mother may be reused, both variables are bound. If only the family may be reused, the first variable remains unbound, while the second variable is bound.

5 EVALUATION

In the following, we will first examine the *behavior* of the transformations presented above. Subsequently, we will discuss a number of *issues* which emerged in the development of the solution to the benchmark case.

5.1 Behavior

The following statements refer to the improved version of the solution presented in Section 4.2. More specifically, we will refer to the forward transformation ensuring injective mappings (Section 4.2.1) and the configurable backward transformation avoiding duplicate mappings (Section 4.2.2). Although the backward transformation does not ensure injectivity, it may be easily rewritten into an injective transformation by applying the method demonstrated for the forward transformation. Altogether, the injective forward transformation and the injective and configurable backward transformation constitute a *best effort solution* in the sense that its functional behavior cannot be improved further if QVT-R is used as the transformation language (assuming the semantics defined in the standard).

5.1.1 Batch Transformations

The solution satisfies all requirements concerning batch transformations (Section 3.2). This is illustrated in Figure 3, in which models are represented by a simple and intuitive textual notation. The family register shown in Figure 3a is transformed into the person register (b), even though the family register contains two family members with the same name (`John Smith`). Furthermore, a batch backward transformation of (b) results in (c) if parents and existing families are preferred, in (d) if parents and new families are preferred, in (e) for preferred children and existing families, and in (f) for preferred children and new families. Notably, (c) depends on the order in which family members are processed. Since there is no way to resolve this non-determinism for batch transformations (please recall that all collections are assumed to be unordered), this behavior does conform to the problem specification.

5.1.2 Incremental Transformations

The requirements to incremental transformations (Section 3.3) are satisfied only partially. Insertions and deletions are propagated correctly in both directions as long as there are no duplicate names (see below). In particular, in backward direction the transformation definition ensures that the configuration parameters take effect only for new persons (e.g., it is ensured that a person who has already been mapped to a parent is not re-mapped to a child).

The problems which are caused by duplicate names are illustrated by Figure 4. Let us assume that the person register (a) has been transformed into the family register (b), with both parents and existing families being preferred. During this transformation, a *temporary trace* is constructed which records the mappings between family members and persons. This trace is thrown away after the transformation; thus, the correspondences are lost. Next, one of the persons is deleted (c). When propagating this change, we would like to ensure that the corresponding family member is deleted. However, it can only be guaranteed that a corresponding family member is deleted, resulting either in (d) or (e). Thus, propagation is *non-deterministic*. A *deterministic* behavior could be achieved only with the help of a *persistent trace*.

```
a)
Family Register
  Family Smith
    Father John
    Mother Mary
    Son Kevin
    Daughter Katie
  Family Smith
    Father John
    Mother Claire
  Family Miller
```

```
b)
Person Register
  Male Smith, John
  Female Smith, Mary
  Male Smith, Kevin
  Female Smith, Katie
  Male Smith, John
  Female Smith, Claire
```

```
c)
Family Register
  Family Smith
    Father John
    Mother Mary
    Son Kevin
    Son John
    Daughter Katie
    Daughter Claire
```

```
d)
Family Register
  Family Smith
    Father John
  Family Smith
    Mother Mary
  Family Smith
    Father Kevin
  Family Smith
    Mother Katie
  Family Smith
    Father John
  Family Smith
    Mother Claire
```

```
e)
Family Register
  Family Smith
    Son John
    Son Kevin
    Son John
    Daughter Mary
    Daughter Katie
    Daughter Claire
```

```
f)
Family Register
  Family Smith
    Son John
  Family Smith
    Daughter Mary
  Family Smith
    Son Kevin
  Family Smith
    Daughter Katie
  Family Smith
    Son John
  Family Smith
    Daughter Claire
```

Figure 3: Test cases for batch transformations.

Furthermore, there is no way to recognize name changes or moves. This is illustrated in Figure 5 for name changes. Initially, the family register (a) has been transformed into the person register (b). Concurrently, the first name is changed in the family register (c), and the birthday is set in the person register (d). Transforming incrementally in forward direction results in the person register (e), where `Mary` has been deleted and `Mary-Anne` has been re-inserted (with an unset birthday). Thus, name changes result in deletions and insertions. This behavior is caused by the check-before-enforce semantics: Since there is no matching target pattern any more, the old person is deleted. Furthermore, a new person is inserted with the matching full name. There is no way to "fix" an existing target pattern instance such it matches again a modified source pattern instance.

Finally, it is not possible to propagate changes in the order in which they occurred. This is illustrated in Figure 6. Let us assume that an empty person register (a) has been transformed into an empty family register (b). For further propagations, both parents and families are preferred. In the *delta* (sequence of change operations) shown in (c), `John` is inserted before `Jack`. An *order-preserving propagation* of these changes results in (d). If changes are applied in the opposite order (e), order-preserving propagation results in (f), which differs from (d). Since both deltas result in the same person register (g), the QVT-R engine cannot reconstruct the delta and returns either (d) or (f), depending on the non-deterministic selection of the order in which source pattern instances are processed.

```
a)
Person Register
  Male Smith, John
  Male Smith, John
```

```
b)
Family Register
  Family Smith
    Father John
    Son John
```

```
c)
Person Register
  Male Smith, John
```

```
d)
Family Register
  Family Smith
    Father John
```

```
e)
Family Register
  Family Smith
    Son John
```

Figure 4: Test case for duplicates.

```
a)
Family Register
  Family Smith
    Mother Mary
```

```
b)
Person Register
  Female Smith, Mary
    Birthday = undefined
```

```
c)
Family Register
  Family Smith
    Mother Mary-Anne
```

```
d)
Person Register
  Female Smith, Mary
    Birthday = 1980-05-07
```

```
e)
Person Register
  Female Smith, Mary-Anne
    Birthday = undefined
```

Figure 5: Test case for name changes.

```
a)
Person Register
```

```
b)
Family Register
```

```
c)
1 : Insert Male
      ("Smith, John")
2 : Insert Male
      ("Smith, Jack")
```

```
d)
Family Register
  Family Smith
    Father John
    Son Jack
```

```
e)
1 : Insert Male
      ("Smith, Jack")
2 : Insert Male
      ("Smith, John")
```

```
f)
Family Register
  Family Smith
    Father Jack
    Son John
```

```
g)
Person Register
  Male Smith, John
  Male Smith, Jack
```

Figure 6: Test case for order-dependent changes.

5.1.3 Transformation Laws

In Section 3.4, we introduced the notions of correctness and hippocraticness, as well as round-trip laws. Correctness has already been discussed above. Both forward and backward transformations are *hippocratic*: If the transformation is re-executed in the same direction, the target model is not modified any more. Please notice that this property holds even if configuration parameters are changed for the backward transformation: The parameters affect only the transformation of new persons. Since no changes are applied after the first run, there are no new persons; rather, all persons already have matching family members being reused in the transformation.

Likewise, *round-trip laws* hold for both transformation orders (forward-backward and vice versa). For example, after having transformed the family register in Figure 3a into the person register (b), an immediate incremental transformation in the opposite direction does not modify the original family register any more — even though this state of the families model could not have been created by any batch transformation.

However, it should be noted that in general neither hippocraticness nor round-trip laws are guaranteed *per se*: In QVT-R, transformations may be defined which violate these laws. Thus, transformation laws need to be checked for each transformation definition.

5.2 Issues

In the case study presented in this paper, we identified the following issues summarized below.

Issue 1 (Imprecise change propagation)**.** *Due to the strictly state-based design, changes may be propagated only imprecisely.*

As already mentioned in Section 2, QVT-R follows a purely *state-based approach* to incremental transformations: The only information which is assumed to be present are the states of the source and the target model. In this way, the prerequisites for the execution of incremental transformations are minimized. In particular, QVT-R assumes neither deltas nor persistent traces. As shown in Section 5.1.2, this decision limits the preciseness of change propagations: In the presence of duplicates, precise change propagation requires persistent traces. For example, if one of two family members with the same name is deleted, it is impossible to guarantee deletion of "the corresponding person" because the correspondences are not stored. Furthermore, order-dependent changes can be supported only with the help of deltas. Finally, changes in source pattern instances cannot be propagated to target pattern instances, due to the check-before-enforce semantics. Thus, name changes and moves cannot be propagated.

Issue 2 (Unidirectional transformations)**.** *Bidirectional transformation problems may have to be solved by pairs of unidirectional transformation definitions.*

The Familes to Persons case demands for bidirectional transformations. QVT-R supports the definition of both uni- and bidirectional transformations. While we proposed a bidirectional transformation definition as an initial solution (Section 4.1), the improved solution required two unidirectional transformation definitions because the rules for forward and backward transformations differ significantly (Section 4.2). First, the approach for ensuring injective mappings requires a unidirectional transformation definition, due to the bookkeeping of matches in the source and the target model. Second, the backward transformation needs to resolve nondeterminism in a configurable way; the respective rules have no counterpart in the definition of the forward transformation.

Issue 3 (Non-injective mappings)**.** *Due to the check-before-enforce semantics, multiple source pattern instances may be mapped to the same target pattern instance.*

The Families to Persons case is designed judiciously such that keys may be assumed neither in the families nor in the persons model. Thus, the check-before-enforce semantics results in a reuse of a person if a person with the same name is already present (likewise in the opposite direction). Nevertheless, we managed to present a solution which enforces injective mappings (Section 4.2.1). However, this solution requires to keep track of matches in both the source and the target model and has a strongly procedural flavor. Furthermore, it excludes the development of a single bidirectional transformation definition.

Issue 4 (Duplicate transformation)**.** *Since relations are applied independently of each other, model elements may be transformed multiple times.*

In QVT-R, relations are applied to source pattern instances when they satisfy the constraints defined in the relation. Thus, the same source pattern instances may be transformed multiple times. This problem occurred in the initial version of the transformation definition in backward direction (a person is mapped both to a parent and a child, see Section 4.1). We solved this problem by rewriting the relations such that each person is considered only once for being transformed. This problem is not uncommon in rule-based languages, but there are other approaches where conflicts are recognized and resolved during execution (Becker et al., 2007).

6 RELATED WORK

Research on QVT-R has primarily focused on theoretical work concerning the semantics definition (Stevens, 2010). For example, (Bradfield and Stevens, 2012) and (Bradfield and Stevens, 2013) address the formalization of check-only and enforcing transformations, respectively. Deviating from the standard, (Macedo and Cunha, 2016) proposes a semantics definition following the principle of least change. However, this proposal retains the restrictions resulting from the state-based approach to change propagation.

A small number of application-oriented papers aim at evaluating different aspects of QVT-R such as expressiveness and conciseness. So far, all of these studies have dealt with bidirectional batch (rather than incremental) transformations (Westfechtel, 2016a; Westfechtel, 2015; Westfechtel, 2016b).

A precursor of the solution to the Families to Persons benchmark presented in this paper was developed for the Transformation Tool Contest (Anjorin et al., 2017a), based on the Benchmarx framework proposed in (Anjorin et al., 2017b). This preliminary solution was provided as an illustrative reference to case developers, but it was not published and did not take part in the contest. Furthermore, the preliminary solution suffered from several limitations (e.g., non-injective mappings) which were addressed in the improved solution presented in this paper.

All solutions were developed and tested with medini QVT — to the best of our knowledge the only tool for QVT-R which is currently available (ikv++ technologies, 2017). While medini QVT conforms to the syntax of the QVT-R standard, it deviates considerably from the semantics defined in the standard. To avoid the discussion of tool-specific behavior, the current paper exclusively uses the semantics definition in the standard as reference point; medini QVT was employed primarily for syntax checking.

7 CONCLUSION

We presented a solution to the Families to Persons case in QVT-R, a declarative bidirectional model transformation language defined as a standard by the Object Management Group. The solution constitutes a best effort approach in the sense that all requirements from the Families to Persons case were addressed which can be addressed in QVT-R. An evaluation of the solution identified several issues such as imprecise change propagation, the need of providing a pair of unidirectional transformation definitions, $n:1$ mappings, and duplicate transformations. These observations motivate further studies concerning the expressiveness of QVT-R with respect to the definition of bidirectional incremental transformations.

Furthermore, future work will address a detailed analysis of the solutions to the Families to Persons case, as described briefly in (Anjorin et al., 2017b) and the papers accepted for the Tool Transformation Contest (Garcia-Dominguez et al., 2017). All of this work is based on the Benchmarx framework, which we consider the first operational framework for evaluating bidirectional transformations. In addition, we are implementing additional cases in the Benchmarx framework, based on the cases proposed in (Westfechtel, 2016a). In this way, we hope to trigger more work on the evaluation of bidirectional transformations — which, as we believe, is urgently needed.

ACKNOWLEDGEMENTS

The author gratefully acknowledges the support provided by Anthony Anjorin and Erhan Leblebici for the integration of the medini QVT implementation of the Families to Persons benchmark into the Benchmarx framework.

REFERENCES

Anjorin, A., Buchmann, T., and Westfechtel, B. (2017a). The families to persons case. In (Garcia-Dominguez et al., 2017).

Anjorin, A., Cunha, A., Giese, H., Hermann, F., Rensink, A., and Schürr, A. (2014). BenchmarX. In Candan, K. S., Amer-Yahia, S., Schweikardt, N., Christophides, V., and Leroy, V., editors, *Workshop Proceedings of the EDBT/ICDT 2014 Joint Conference*, volume 1133 of *CEUR Workshop Proceedings*, pages 82–86, Athens, Greece.

Anjorin, A., Diskin, Z., Jouault, F., Ko, H.-S., Leblebici, E., and Westfechtel, B. (2017b). Benchmarx reloaded: A practical framework for bidirectional transformations. In Eramo, R. and Johnson, M., editors, *Sixth International Workshop on Bidirectional Transformations (BX 2017)*, volume 1827 of *CEUR Workshop Proceedings*, pages 15–30, Uppsala, Sweden.

Becker, S. M., Herold, S., Lohmann, S., and Westfechtel, B. (2007). A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. *Software and Systems Modeling*, 6(3):287–316.

Bradfield, J. and Stevens, P. (2012). Recursive checkonly QVT-R transformations with general when and where clauses via the modal mu calculus. In de Lara, J. and Zisman, A., editors, *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*, volume 7212 of *Lecture Notes of Computer Science*, pages 194–208, Tallinn, Estonia. Springer-Verlag.

Bradfield, J. and Stevens, P. (2013). Enforcing QVT-R with mu-calculus and games. In Cortellessa, V. and Varró, D., editors, *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013)*, volume 7793 of *Lecture Notes of Computer Science*, pages 282–296, Rome, Italy. Springer-Verlag.

Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F. (2009). Bidirectional transformations: A cross-discipline perspective. In Paige, R. F., editor, *Proceedings of the Second International Conference on Theory and Practice of Model Transformations (ICMT 2009)*, volume 5563 of *Lecture Notes of Computer Science*, pages 260–283, Zurich, Switzerland. Springer-Verlag.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17:1–17:65.

Garcia-Dominguez, A., Hinkel, G., and Krikava, F., editors (2017). *Proceedings of the 10th Transformation Tool Contest (TTC 2017)*, volume 2026 of *CEUR Workshop Proceedings*, Marburg, Germany.

Hidaka, S., Tisi, M., Cabot, J., and Hu, Z. (2016). Feature-based classification of bidirectional transformation approaches. *Software and Systems Modeling*, 15(3):907–928.

ikv++ technologies (2017). *medini QVT*. http://projects.ikv.de/qvt.

Libkin, L. (2004). *Elements of Finite Model Theory*. Springer-Verlag, Berlin.

Macedo, N. and Cunha, A. (2016). Least-change bidirectional model transformation with QVT-R and ATL. *Software and Systems Modeling*, 15(3):783–810.

Object Management Group (2014). *Object Constraint Language Version 2.4*. Needham, MA, formal/2014-02-03 edition.

Object Management Group (2016a). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.3*. Needham, MA, formal/2016-06-03 edition.

Object Management Group (2016b). *OMG Meta Object Facility (MOF) Core Specification Version 2.5.1*. Needham, MA, formal/2016-11-01 edition.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2nd edition.

Stevens, P. (2010). Bidirectional model transformations in QVT: Semantic issues and open questions. *Software and Systems Modeling*, 9(1):7–20.

Westfechtel, B. (2015). A case study for evaluating bidirectional transformations in QVT Relations. In Filipe, J. and Maciaszek, L., editors, *Proceedings of the 10th International Conference on the Evaluation of Novel Approaches to Software Engineering (ENASE 2015)*, pages 141–155, Barcelona, Spain. INSTICC, SCITEPRESS.

Westfechtel, B. (2016a). Case-based exploration of bidirectional transformations in QVT relations. *Software and Systems Modeling*. Online First.

Westfechtel, B. (2016b). A case study for a bidirectional transformation between heterogeneous metamodels in QVT Relations. In Filipe, J. and Maciaszek, L., editors, *Proceedings of the 10th International Conference on the Evaluation of Novel Approaches to Software Engineering (ENASE 2015) — Revised Selected Papers*, volume 599 of *Communications in Computer and Information Science*, pages 141–161, Berlin, Heidelberg, New York. Springer-Verlag.