Bidirectional Model Transformations With FunnyQT

Tassilo Horn
horn@uni-koblenz.de
Institute for Software Technology
University of Koblenz-Landau, Germany

Abstract. FunnyQT is a model querying and transformation library. Instead of inventing a new syntax and semantics from scratch, it is designed as an extensible API and embedded DSL in the functional, JVM-based Lisp dialect Clojure.
FunnyQT defines several sub-APIs, each offering constructs for a particular querying or transformation use-case. In this paper, the funnyqt.bidi API is discussed which allows to define bidirectional transformations in the style of the QVT Relations language.

1 Introduction

This paper introduces the bidirectional transformation API of the model querying and transformation library FunnyQT[8]. FunnyQT is not a new language with its own syntax and semantics, but it is designed as an API in the functional, JVM-based Lisp dialect Clojure[2].

FunnyQT provides several namespaces (APIs), each dedicated to some concrete querying or transformation task. For example, there is a namespace dedicated to pattern matching, a namespace dedicated to in-place transformations, a namespace dedicated to unidirectional model transformations, a namespace dedicated to bidirectional transformations, and several more.

As a Lisp dialect, Clojure provides strong metaprogramming capabilities in terms of macros [7,9]. A macro is a kind of function that is called at compile-time. It receives code via its arguments, and it returns code that takes the place of its call. Since Clojure code is represented using Clojure datatypes itself (literals, symbols, lists, vectors, maps), a macro is in essence a transformation on the abstract syntax tree of the code passed to the macro. FunnyQT makes extensive use of these capabilities to create task-oriented embedded or internal DSLs [4].

FunnyQT is able to work on EMF models (Eclipse Modeling Framework, [11]) and JGraLab TGraphs[3]. The transformation example discussed in Section 3 transforms and synchronizes between EMF models, but it is perfectly fine to have transformations between models of different modeling frameworks, too.

1 http://jgralab.github.io/funnyqt/
2 http://www.clojure.org
3 http://jgralab.uni-koblenz.de
For achieving bidirectionality in model transformations, FunnyQT’s bidirectional transformation API `funnyqt.bidi` builds upon its relational model querying API `funnyqt.relation` which is in turn based on the `core.logic` library that implements relational programming capabilities for Clojure. The most important concepts of relational programming including FunnyQT’s extensions for relational model querying are discussed in Section 2. Section 3 then introduces FunnyQT’s bidirectional transformation API using a simple example. Section 4 discusses related transformation approaches, and Section 5 concludes the paper.

2 Relational Querying

The `core.logic` library is a Clojure port of `miniKanren` [3] which embeds Prolog-style relational programming into the functional language Scheme. It also implements cKanren [1], an extension to miniKanren for constraint logic programming over trees (CLP(Tree)) and finite domains and (CLP(FD)). The latter allows for defining arithmetical constraints over integral values.

As said, `core.logic` and `miniKanren` embed relational programming inside their functional host languages Clojure and Scheme, i.e., there is a macro `run*` inside which code is written that is evaluated according to relational semantics. An example relational program is given in the following listing. In there and in the rest of the paper, all `core.logic` constructs are prefixed with the namespace alias `ccl`, and constructs of its finite domain extension are prefixed with `cclfd`.

```clojure
1 (ccl/run* [q]
2  (ccl/fresh [a b c]
3   (ccl/== q [a b c])
4   (cclfd/in c (cclfd/interval 3 5))
5   (ccl/membero i q)
6   (ccl/conde
7     [(ccl/conso a [2 3] q)]
8     [(ccl/resto q [1 c]) (ccl/== a 3)])
9  )
10 )
```

The argument to `run*` is a logic variable `q`. The `run*` semantics is to compute all possible values of `q` that are solutions to all the constraints imposed in `run*`’s body (lines 2-9).

`fresh` introduces `fresh` logic variables. A logic variable can either be `ground` (bound to a value) or `fresh` (not bound to a value). All other constructs, i.e, `==` (unify), `membero` etc., are `relations`.

The application of a relation to logic variables or literals, e.g., `(ccl/== q [a b c])`, is called a `goal`. Goals emit candidate bindings for the logic variables passed to them, and the logic engine’s task is to find those bindings for which all goals are satisfied.

4 [http://github.com/clojure/core.logic](http://github.com/clojure/core.logic)
5 [http://miniKanren.org](http://miniKanren.org)
6 Scheme doesn’t have namespaces, so relations are suffixed (usually with an “o”) to prevent name clashes with standard functions. Clojure doesn’t have this problem, nevertheless, this convention is frequently followed.
Line 3 unifies \( q \) with a vector of three elements \( a, b, \) and \( c \). Line 4 imposes a finite domain constraint: the value of \( c \) has to be an integer in the interval from 3 to 5. Line 5 says that 1 must be a member of \( q \). The `conde` in line 6 defines a disjunction of two alternatives. Each alternative is a conjunction of goals. The first one in line 7 specifies that consing (adding in front) \( a \) onto \([2 3]\) results in \( q \). Clearly, in that case, \( a \) must be 1 in order to make line 5 succeed, and \( b \) and \( c \) are 2 and 3, respectively. The second alternative in line 8 says that the rest (i.e., the tail) of \( q \) is \([1 3]\) implying \( b \) to be 1, and \( a \) is 3. \( c \) isn’t constrained any further, so it may take any value as defined by line 4. All in all, the relational program returns the solutions printed in line 9.

### 2.1 Relational Model Querying

The core.logic library is extensible, and FunnyQT’s `funnyqt.relational.emf` namespace defines several relations for querying EMF models. In the following examples, this namespace is bound to the alias `remf`.

- `(eobjecto m eo)` is a relation where \( eo \) is an element of model \( m \).
- `(eobjecto m eo t)` is a relation where \( eo \) is an element of model \( m \) which has the metamodel type \( t \).
- `(valueo m eo attr val)` is a relation where \( val \) is the value of \( eo \)’s \( attr \) attribute.
- `(adjo m eo ref reo)` is a relation where \( eo \) references \( reo \) with its \( ref \) reference.

For these relations, the first argument, the model \( m \), must always be ground, but any other argument may be fresh.

For illustrating the introduced model querying relations, the EMF model shown in Figure 1 is used. It is bound to the variable \( fm \) in the next listings. This model conforms to the simple family metamodel (borrowed from the ATL tutorial) shown in Figure 2.

As an example, the program in the next listing poses the question: What is the first name (\( \text{fname} \)) of the \text{Member} element \( \text{obj} \) which has some attribute \( \text{attr} \) set to 36, and what is the name of this attribute?

```clike
1 (ccl/run* [q])
2  (ccl/fresh [obj fname attr])
3   (remf/eobjecto fm obj "Member")
4   (remf/valueo fm obj :firstName fname)
5   (remf/valueo fm obj attr 36)
6   (ccl/== q [fname attr]))
7 ;=> ("Stella" :age)
```

The symbol `Member` in line 3 is *quoted* (‘) in order to prevent it from variable resolution.FunnyQT uses quoted symbols to denote type names. Identifiers prefixed with a colon such as `:firstName` are *keywords*. A keyword always evaluates to itself. FunnyQT uses keywords to denote attribute and reference names.

Line 3 specifies that \( \text{obj} \) must be a `Member`. In line 4, the logic variable `fname` is unified with the value of \( \text{obj} \)’s `firstName` attribute. Line 5 then specifies that \( \text{obj} \) must also have some attribute whose value is 36, binding `attr` to that attribute’s

---

7 Similar relations exist for TGraphs in `funnyqt.relational.tg`.
name. Line 6 just defines the answers to be tuples of the `fname` and `attr` values. The only solution is Stella, because her `age` attribute has the value 36.

eobjecto, `valueo`, and `adjo` are the low-level relations for querying EMF models relationally. FunnyQT also provides macros for generating convenient, `metamodel-specific relational querying APIs`, and those are used throughout the rest of this paper. For EMF metamodels, the macro’s name is `generate-ecore-model-relations`. It is applied as shown in the next listing.

```
(remf/generate-ecore-model-relations "Families.ecore" icmt14-funnyqt.families f)
```

The macro receives the metamodel file, a symbol denoting the namespace in which the relations should be generated, and the alias `f` under which that namespace can be accessed from the current namespace. For the metamodel in Figure 2, the following relations are generated:

- `(Member m o)` is an `element type relation` where `o` is a `Member` element of model `m`. Analogous relations are generated for the classes `FamilyModel` and `Family`.  

![Fig. 1. A simple family model conforming to the metamodel in Figure 2. The members links from the `FamilyModel` to all `Member` objects are not printed for layout reasons.](image1)

![Fig. 2. A simple EMF family metamodel](image2)
(firstName m o fn) is an attribute relation where fn is the value of o’s firstName attribute, and o is an element of model m. Analogous relations are generated for the attributes lastName and age.

(->daughters m f d) is a reference relation where the element f of model m references d with its daughters reference. Analogous relations are generated for the references father, mother, sons, and for the corresponding opposite references.

Since property names don’t need to be unique, the attribute and reference relations consider objects of all classes that do have an attribute/a reference of that name.

Custom relations can be defined as ordinary Clojure functions using defn. For example, fathero shown in the following listing is a relation where father is the father of child in the family model fm.

```
(defn fathero [fm father child]
  (ccl/fresh [family]
    (f/->father fm family father)
    (ccl/conde
     [[(f/->daughters fm family child)]
      [(f/->sons fm family child)])
     )))
```

This new relation can be used like any of the generated relations, e.g., to query for all fathers older than 40 years together with their children.

```
(ccl/run* [q]
  (ccl/fresh [father fname fage child cname]
    (fathero fm father child)
    (f/firstName fm father fname)
    (f/age fm father fage)
    (cc1f/-> fage 40)
    (f/firstNStle fm child name)
    (ccl/== q [fname cname]))
  ;=> (["Steve" "Stella"] ["Steve" "Stu"] ["Steve" "Jim"] ["Steve" "Dennis"]
   ; ["Chris" "Debby"] ["Chris" "Carol"] ["Chris" "Conzuela"])
```

There’s one last custom relation being introduced here in the next listing which is used by the example transformation in Section 3.1. lastnameo is a relation where member is a Member element in model fm and lastname is his or her last name.

```
(defn lastnameo [fm member lastname]
  (ccl/fresh [family]
    (ccl/conde
     [[(f/->father fm family member) (f/lastName fm family lastname)]
      [(f/->mother fm family member) (f/lastName fm family lastname)]
      [(f/->daughters fm family member) (f/lastName fm family lastname)]
      [(f/->sons fm family member) (f/lastName fm family lastname)])
     )))
```

Every member can be father or mother of some family and son or daughter of some other family, but for the last name, the family where the member is in a parental role has to be preferred. This is done using conde which is a disjunction of alternatives like conde. Every alternative is tested one after the other, but as soon as one succeeds, the remaining ones are ignored. This is similar to Prolog’s cut-operator. With reference to Figure 1 when member refers to Debby, lastname can only be Smith. With conde, there would be another solution where lastname was Carter.

The core.logic based relational model querying API discussed in this section is the foundation for the FunnyQT bidirectional transformation API which is discussed in the next section.
3 Bidirectional Transformations

In FunnyQT, a bidirectional transformation specifies correspondences between two models, one called the left model and one called the right model. For that purpose, it consists of several transformation relations, each defining a correspondence between elements in the left and elements in the right model. To disambiguate those from core.logic relations, we call them t-relations in this paper. Such a t-relation has the following form, where when-goals, left-goals, and right-goals are all conjunctions of arbitrarily many core.logic goals.

```<t-relation-name>
:when [<when-goals>]
:left [<left-goals>]
:right [<right-goals>]
:where [<t-relation-calls>])
```

Every t-relation relates elements in the left model with elements in the right model by unifying over logic variables that occur in both the left-goals and the right-goals and express the correlation based on property values. In the simplest case, if a logic variable denotes an attribute value of an element in the left model and also an attribute value of an element in the right model, then these two elements must have the same value set for the respective attributes.

A t-relation may have a precondition (the :when clause) usually used to refer to elements already related by other t-relations, and it may have a :where clause specifying t-relations that also need to hold for the elements related by the current t-relation. Frequently transformations are designed along the aggregation hierarchy of the models. That is, there is one top-level t-relation relating the models’ root elements, and that t-relation calls other t-relations with its :where clause which relate the respective child elements, and so forth.

A transformation is always executed (enforced) in a given target direction, either left or right. The t-relation semantics when transforming in the direction of the right model is as follows: For all possible logic variable bindings satisfying the conjunction of when-goals and left-goals, there has to be at least one binding of the remaining fresh variables satisfying the right-goals. Furthermore, the t-relations given in the :where clause must be enforced. Note that those are not enforced before all correspondences of the current t-relation have been established, i.e., the evaluation order is breadth-first rather than depth-first.

The conjunction of goals of the :when and the respective source direction clause are evaluated like discussed in the model querying examples given in Section 2.1.

The respective target direction clause, however, is evaluated in a special enforcement mode. In this mode, the element type relations emit as candidate bindings for the logic variables both wrappers for existing target model elements of that type and temporary placeholder elements. The attribute and reference relations add further information about the required property values of candidate elements. Both wrappers and temporary elements can be manifested (i.e., updated or created) in the target model to make the t-relation hold.
FunnyQT then selects the first candidate binding that satisfies all goals and manifests it. By construction of the underlying algorithm, the sequence of candidate bindings is ordered from bindings with only existing elements to bindings with only temporary elements, thus no new elements will be created if the goals can be satisfied by elements that already exist in the target model.

The generated attribute and property relations discussed in Section 2.1 have a very narrow update semantics for an existing target model element $el$. Concretely, $(\text{attr} \ m \ el \ \text{val})$ only succeeds if either $el$’s attr value equals val, or if $el$’s attr attribute is not set (in which case manifestation will set it to val). The same holds for single-valued references. This means that during enforcement of a t-relation, if there’s an existing element that satisfies all goals except one single attribute or single-valued reference goal, that element won’t be reused and modified but a new one is created.

In practice, however, most elements can be uniquely identified by a subset of their properties. In that case, a transformation writer should be able to specify that only these properties need to match while other properties might be modified to make the t-relation hold. Therefore, the generated metamodel-specific relational APIs also define relations $(\text{attr}^* \ m \ el \ \text{val})$ for any attribute attr and relations $(\rightarrow\text{ref}^* \ m \ el \ \text{trg})$ for any single-valued reference ref. They act exactly as their unstarred counterparts when used as a query, but in enforcement mode they allow the modification of $el$’s current value.

All bindings of the logic variables in the :when, :left, and :right clause are saved as traceability mapping that can be accessed by other t-relations using a predefined FunnyQT relation relateo. Furthermore, the complete traceability mappings of all t-relations are the result of a bidirectional transformation, the modification of the enforced target model being only a side-effect.

### 3.1 A Transformation Example

In this section, the explanations given above are exemplified using a concrete transformation example. The transformation expects family models conforming to the metamodel shown in Figure 2 as left models, and genealogy models conforming to the metamodel shown in Figure 3 as right model.

$$\text{Fig. 3. The EMF genealogy metamodel}$$

Like for the family metamodel, we generate a metamodel-specific relation API for the genealogy metamodel. The generated API is made accessible via the namespace alias $g$. 
The generated relational genealogy API consists of (a) the element type relations \texttt{Genealogy, Person, Male,} and \texttt{Female}, (b) the attribute relations \texttt{fullName} and \texttt{yearOfBirth} and their modification allowing versions \texttt{fullName*} and \texttt{yearOfBirth*}, and (c) the reference relations \texttt{->persons, ->parents, ->children, ->husband, and ->wife} plus the modification allowing versions \texttt{->husband*} and \texttt{->wife*}.

In the following, the complete transformation relating family models to genealogies is discussed. Table 1 gives an overview of the used namespaces and which alias is used to refer to them.

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clojure.core.logic</td>
<td>ccl</td>
<td>General relations</td>
</tr>
<tr>
<td>clojure.core.logic</td>
<td>cclfd</td>
<td>Constraints over finite domains</td>
</tr>
<tr>
<td>funnyqt.bidi</td>
<td>bidi</td>
<td>FunnyQT bidirectional transformation API</td>
</tr>
<tr>
<td>funnyqt.relational</td>
<td>r</td>
<td>FunnyQT relational querying API</td>
</tr>
<tr>
<td>funnyqt.relational.emf</td>
<td>remf</td>
<td>FunnyQT relational querying API (EMF)</td>
</tr>
<tr>
<td>icmt2014-funnyqt.families</td>
<td>f</td>
<td>Generated metamodel API (Figure 2)</td>
</tr>
<tr>
<td>icmt2014-funnyqt.genealogy</td>
<td>g</td>
<td>Generated metamodel API (Figure 3)</td>
</tr>
</tbody>
</table>

Table 1. Namespaces used by the transformation

\textit{The Transformation Definition.} A bidirectional FunnyQT transformation is defined using the \texttt{deftransformation} macro. It receives a symbol denoting the transformation’s name, a vector of parameters denoting the left and the right model plus possibly additional parameters, and arbitrary many t-relations.

The example transformation \texttt{families2genealogy} is defined as follows.

1. (bidi/deftransformation families2genealogy [[fm gm] year]
2. ;; t-relations...
3. )

The left family model will be bound to the parameter \texttt{fm}, the right genealogy model will be bound to \texttt{gm}, and there is one additional parameter \texttt{year} denoting the current year which is needed for converting between ages and years of birth. All following listings contain the t-relations contained by the transformation.

\textit{T-Relations.} The first t-relation is \texttt{familymodel2genealogy} shown in in the next listing.

1. (:top familymodel2genealogy
2. :left [[f/FamilyModel fm ?family-model]]
3. :right [[g/Genealogy gm ?genealogy]]
4. :where [[f/family2parents :?model ?family-model :?genealogy ?genealogy]]

The \texttt{:^:top} metadata annotation specifies that it is a top-level t-relation which is called by the transformation engine automatically. If there are more than one top-level t-relations, they are enforced in the order of their declaration.
In t-relations, variables prefixed with a question mark denote logic variables. The t-relation specifies that for all FamilyModel elements \(?family-model\) in the left model \(fm\), there has to be a Genealogy element \(?genealogy\) in the right model \(gm\), and vice versa.

The :where clause specifies that for all possible values of \(?family-model\) and \(?genealogy\), the t-relation \(family2parents\) has to hold. Every t-relation has an implicit named parameter (in contrast to positional parameters) for each of its logic variables. So the argument syntax \(:?fmodel \ ?family-model\) means that the logic variable \(?fmodel\) inside \(family2parents\) should be bound to the current value of \(?family-model\). Thus, \(?fmodel\) and \(?genealogy\) are already ground in \(family2parents\) when being called from this :where clause.

The t-relation \(family2parents\) relating a Family with father and mother Member elements in the left model to a Male and a Female element in the role of a husband and his wife in the right model is shown in the next listing.

```
(family2parents
 :left [(f/->families fm ?fmodel ?family)
 (f/Family fm ?family)
 (f/lastName fm ?family ?family-lname)
 (f/->father fm ?family ?father-member)
 (f/->members fm ?fmodel ?father-member)
 (f/Member fm ?father-member)
 (f/lastName fm ?father-member ?father-lname)
 (f/age* fm ?father-member ?father-age)
 (cclfd/- year ?father-age ?husband-yob)
 (f/->mother fm ?family ?mother-member)
 (f/->members fm ?fmodel ?mother-member)
 (f/Member fm ?mother-member)
 (f/firstName fm ?mother-member ?mother-fname)
 (f/age* fm ?mother-member ?mother-age)
 (cclfd/- year ?mother-age ?wife-yob)
]:right [(g/->persons gm ?genealogy ?husband)
 (g/Male gm ?husband)
 (g/fullName gm ?husband-fullname)
 (g/yearOfBirth* gm ?husband ?husband-yob)
 (g/->persons gm ?genealogy ?wife)
 (g/Female gm ?wife)
 (g/fullName gm ?wife-fullname)
 (g/yearOfBirth* gm ?wife ?wife-yob)
 (r/stro ?father-fname " " ?family-lname ?husband-fullname)
 (r/stro ?mother-fname " " ?family-lname ?wife-fullname)
 (g/->wife gm ?husband ?wife)])
```

The correspondences between elements in the left and elements in the right model are established by constraints on attribute values.

1. In the left family model, members have a \(firstName\), and the \(lastName\) is stored with their family. In the right genealogy model, persons have \(fullName\). The goal in line 30 specify that concatenating the \(?father-member\’s first name with his \(?family\’s last name gives the full name of the corresponding \(Male\ element \(?husband\) in the right model. The goal in line 31 does the same for the left \(?mother-member\) and the right \(?wife\).

2. Member elements in the left model have an \(age\) whereas right Person elements have a \(yearOfBirth\). Line 15 says that when substracting the \(?father-age\) from the current \(year\), one gets the year of birth (\(?husband-yob\)) of the corresponding Male element \(?husband\) in the right model. Line 21 says the same for the \(?mother-member\) and the corresponding Female element \(?wife\).
The correlation imposed by the t-relation is also visualized in Figure 4.

Fig. 4. A visualization of the family2parents t-relation

Since in our case we want to distinguish persons by their name and their relationships between each other, we use the non-modifying relations for the firstName, lastName, and fullName attributes, and for the single-valued wife reference. For the age and year of birth, however, we use the relations age* and yearOfBirth* which allow modification.

So far, the t-relations ensure that the left model contains a FamilyModel root element corresponding to a Genealogy root element in the right model, and that for each Family with father and mother in the left model, there is a corresponding couple consisting of a male and a female in the right model, and vice versa.

Abstract T-Relations and T-Relation Specialization. Up to now, the t-relations do not apply to members or persons that aren’t in a parental role. Thus, t-relations relating sons to male children, and daughters to female children are needed. Clearly, both t-relations will be very similar. To cope with situations like this, FunnyQT has the concept of abstract t-relations which cannot be enforced on their own but can be extended by other t-relations.

Extension means that extending t-relations implicitly include the extended relations’ :when, :left, :right, and :where clauses in their own respective clauses (possibly with some logic variables renamed).

The following listing defines an abstract t-relation child2person, relating a left ?child-member to a right ?person.

```
(abstract child2person
  :when [(bi/di/relation family2parents :?model :?model
         [:genealogy ?genealogy :?family :?family
          :husband ?father :?wife ?mother])
  :left [(f/Member fm ?child-member)
         (f/->members fm :?model ?child-member)
         (f/firstName fm ?child-member ?firstname)
         (lastnameo fm ?child-member ?lastname)
         (f/age* fm ?child-member ?age)
         (cclfd/- year ?age ?yob)]
  :right [(g/->persons gm ?genealogy ?person)
           (g/Person gm ?person)
           (g/->children gm ?father ?person)
```

child2person uses the :when clause to refer to already established correspondences of family2parents using the predefined FunnyQT relation relatoo. Every quintuple consisting of ?fmodel, ?genealogy, ?family, ?husband, and ?wife that have already been related by family2parents is bound to ?fmodel, ?genealogy, ?family, ?father, and ?mother in child2person.

The :left and :right clauses then relate a member ?child-member contained in ?fmodel to a person ?person in the ?children role of both ?father and ?mother. Again, concatenating the left ?child-member’s first name, a space, and the ?family’s last name gives the full name of the corresponding child ?person in the right model. However, here the lastnameo relation discussed in Section 2.1 is used in order to take the correct family’s lastName value.

The abstract child2person t-relation is then extended by two concrete t-relations son2male and daughter2female as shown in the following listing. These are top-level t-relations again, so they are called after familymodel2genealogy (which also enforces family2parents) already holds.

- son2male extends child2person. The arguments specify logic variable renamings. What is called ?child-member in child2person is called ?son-member in son2male, and child2person’s ?person is called ?son in son2male. So extending t-relations may use different names for the same elements and values as the t-relations that are extended.

At compile-time, the transformation definition is expanded to a plain Clojure function families2genealogy which takes four parameters: the left model, the right model, the transformation direction (either :left or :right), and the additional parameter year declared by the transformation. The first three parameters are the same for all bidirectional FunnyQT transformations.

When transforming the family model in Figure 1 in the direction of an empty genealogy model gm using (families2genealogy fm gm :right 2014), the populated
result model is shown in Figure 5 (which excludes the root Genealogy element for layout reasons). When transforming this model in the direction of an empty family model using \( \text{families2genealogy fm gm :left 2014} \), the populated result model exactly equals the model in Figure 1. Synchronizing between those two models in either direction doesn’t change anything in the respective target model, because all t-relations already hold.

If we would add a new Female element as a wife to the currently unmarried Jim Smith in Figure 5 and transformed back into the direction of the left family model, this would create a new Family with last name Smith where the (existing) Member Jim is in the father role and a new Member for Jim’s wife is in the mother role.

If we would not only capture the year of birth but the complete birth date of persons in the genealogy model, we could also run the transformation in the direction of the left model on each birthday of any contained person to keep the values of the age attribute of family members up-to-date.

### 3.2 Characteristics

The literature (e.g., Stevens in [12]), distinguishes several properties such as determinism, correctness, hippocraticness, and undoability that bidirectional transformations may have. Here, we will elaborate which of those apply to FunnyQT.
FunnyQT bidirectional transformations are deterministic, i.e., running a transformation in either direction on the same two models will always result in the same effects.

The definition given in [12] requires correctness in the following sense. Let \( T \subseteq M \times N \) be a relation between models conforming to the metamodels \( M \) and \( N \), and \( T(m, n) \) holds if and only if \( m \) and \( n \) are consistent. Such a consistency relation encodes two directional transformations \( T^{-1} : M \times N \rightarrow N \) and \( T^{-1} : M \times N \rightarrow M \). \( T \) is said to be correct if \( \forall m \in M \forall n \in N : T(m, T^{-1}(m, n)) \land T(T(m, n), n) \) holds.

We doubt that conforming to this definition is preferable in practice as it implies deletion of those elements in the respective target model that have no correspondence in the respective source model. FunnyQT has made the design choice of never deleting elements. Consider the situation of two consistent models that are then extended in parallel. If FunnyQT transformations were correct according to that definition, synchronization between the models would be impossible. However, since FunnyQT transformations return the complete traceability mappings, it is easy to delete elements with no correspondence after the transformation has finished so that the composition of the transformation followed by the cleanup is correct according that definition.

Bidirectional FunnyQT transformations are hippocratic, i.e., if two models are already consistent according to the relation implied by the transformation, transforming in either direction won’t modify the models.

A last property is undoability. Let \( T(m, n) \), i.e., the models \( m \) and \( n \) are consistent. Now, the model \( m \) is modified to \( m' \). \( T \) is undoable if and only if \( T^{-1}(m, T^{-1}(m, n)) = n \) and analogously for \( T^{-1} \) with a modified version \( n' \) of \( n \). For practical reasons, undoability is too strong as a general requirement. But note that with transformations being correct according the definition discussed above, \( T^{-1} \) might delete information which is irreversibly lost at least in non-bijective scenarios. In contrast, because FunnyQT transformations won’t delete elements, the result of \( T^{-1}(m, T^{-1}(m', n)) \) is a model which at least contains \( n \), but which might also contain additional elements with no corresponding elements in \( m \). So we argue that FunnyQT due to its slightly relaxed notion of correctness fosters undoability.

4 Related Work

FunnyQT’s bidirectional transformation API is intentionally designed to be similar to the QVT Relations language (QVTr, [10]). A transformation is decomposed into transformation relations, each relating elements in a left model to elements in a right model. There has to be at least one top-level t-relation that is executed automatically, and a t-relation may enforce other t-relations in terms of a \textit{where} clause. \textit{when} clauses can be used specify that a t-relation only needs to hold for elements that are already related by other t-relations. Also, FunnyQT’s t-relation execution semantics follow QVTr’s \textit{forall-there-exists} approach.
There are some differences between QVTr and FunnyQT’s bidirectional transformation facility, though. QVTr transformations delete target elements that correspond to no source element (according to the correctness definition discussed in Section 3.2) whereas FunnyQT transformations intentionally do not. But because FunnyQT transformations return the complete traceability information, the deletion of elements with no correspondence may be implemented as an additional step if required.

Furthermore, QVTr and FunnyQT have different concepts for specifying which element properties may be overridden. In QVTr, there is the concept of *keys* which allows to specify which subset of properties of a metamodel class should be used to uniquely identify instances of that class. Whenever a QVTr relation doesn’t find a valid target element and a new one is going to be created, it is first checked if there is one where at least the key properties match. If there is, this element is changed to make it match instead of creating a completely new element. FunnyQT, on the other hand, defines separate starred relations for attributes and single-valued references which are allowed to modify their value and which may be used in t-relations. Thus, FunnyQT provides a more fine-granular (t-relation specific) means for allowing property modifications than the transformation-global key concept of QVTr.

The *Janus Transformation Language* (*JTL*, [3]) is similar to FunnyQT bidirectional transformations in that it also uses logic programming, namely *answer set programming* [6], and exploits an existing constraint solver for finding solutions.

Whereas FunnyQT works on the native model representations (EMF or JGraLab) directly, JTL transformations translate the EMF models into an intermediate representation which is then used by the constraint solver, and its results are eventually re-translated back into their original EMF model representation.

The crucial benefit of JTL is that it allows for finding alternative solutions in non-bijective bidirectional transformations scenarios. If there are multiple target models that are consistent with the source model with respect to the transformation rules, JTL is able to enumerate all of them. Also, if a target model is manually changed in a way that it cannot be derived by a forward transformation, then traceability information is employed to propagate back the changes from the modified target model by inferring the closest approximation of an ideal source model.

5 Conclusion and Outlook

In this paper, FunnyQT’s bidirectional transformation API has been discussed which allows for specifying bidirectional model transformations in Clojure. Transformations crossing technological spaces are explicitly supported, and at present there is support for EMF models and JGraLab TGraphs.

A bidirectional transformation consists of transformation relations that relate elements in a left model with elements in a right model. Transformation relations
may be restricted by a precondition, and they may define which other t-relations also need to hold for elements for which the current t-relation holds.

There is also support for multiple inheritance between t-relations which allows for factoring out commonalities of several t-relations in one place. Although not shown in this paper, multiple inheritance between complete transformations is also supported, where the extending transformation may override inherited t-relations and add additional ones.

Although FunnyQT’s bidirectional transformation API is still in an early stage and should not be considered stable yet, its applicability has been tested successfully with several transformation examples, and its expressiveness and means for abstraction such as t-relation and transformation inheritance allow for compact transformation specifications.

In future work, we will consider a visual notion of transformation specifications such as the one shown in Figure 4 for defining bidirectional FunnyQT transformations in order to facilitate the ease of use for non-Clojure programmers.

References