A Petri Net based Debugging Environment for QVT Relations*

Manuel Wimmer*, Gerti Kappel*, Johannes Schoenboeck*, Angelika Kusel†, Werner Retschitzegger‡ and Wieland Schwinger§
*Vienna University of Technology, Business Informatics Group, Austria
Email: lastname@big.tuwien.ac.at
†Johannes Kepler University Linz, Institute of Bioinformatics, Austria
Email: angelika.kusel@bioinf.jku.at
‡University of Vienna, Department of Knowledge and Business Engineering, Austria
Email: werner.retschitzegger@univie.ac.at
§Johannes Kepler University Linz, Department of Telecooperation, Austria
Email: wieland.schwinger@jku.ac.at

***Abstract*—In the Model-Driven Architecture (MDA) paradigm the Query/View/Transformation (QVT) standard plays a vital role for model transformations. Especially the high-level declarative QVT Relations language, however, has not yet gained widespread use in practice. This is not least due to missing tool support in general and inadequate debugging support in particular. Transformation engines interpreting QVT Relations operate on a low level of abstraction, hide the operational semantics of a transformation and scatter metamodels, models, QVT code, and trace information across different artifacts.**

We therefore propose a model-based debugger representing QVT Relations on bases of TROPIC, a model transformation language utilizing a variant of Colored Petri Nets (CPNs). As a prerequisite for convenient debugging, TROPIC provides a homogeneous view on all artifacts of a transformation on basis of a single formalism. Besides that, this formalism also provides a runtime model, thus making the afore hidden operational semantics of the transformation explicit. Using an explicit runtime model allows to employ model-based techniques for debugging, e.g., using the Object Constraint Language (OCL) for simply defining breakpoints and querying the execution state of a transformation.

***Keywords*-QVT Relations; Debugging; Model Transformations; CPN**

I. Introduction

For model transformations which are an essential constituent in MDA [1], the QVT standard [2] proposed by the OMG specifies three different transformation languages: (i) the declarative high-level Relations language, (ii) the declarative low-level Core language, and (iii) the imperative Operational Mapping language. Although tool support and especially debuggers are of utmost importance for declarative languages as stated by Wadler [3], until now, tool support in general and debugging support in particular are still in its infancy [4], [5]. Especially regarding QVT Relations, this seems to be one of the reasons for the lack of adoption in practice. Debugging support is hampered by the following three problems:

(1) QVT Relations code basically consists of declarative correspondence definitions between source and target model elements known as *relations*, which are either directly interpreted or first translated to the QVT Core language and afterwards interpreted by a transformation engine. As a consequence, debugging is limited to the information provided by the interpreter or the transformation engine only, most often consisting of variable values or logging messages, but missing important information, e.g., why a certain relation is executed or at when a certain target element is actually created. Thus, only a snapshot of the actual execution state is provided during debugging while coherence between the specified relations is lost.

(2) As QVT Relations is declarative in nature, the order of rule application needs not to be handled by the transformation designer. Although this relieves transformation designers from a burden when specifying a transformation, the hidden operational semantics is counterproductive for debugging.

(3) Finally, QVT Relations provides a limited view on model transformations, since metamodels, models, QVT code, and trace information are scattered across different artifacts hampering the understanding of the transformation.

We therefore propose a model-based debugger for QVT Relations [6] by employing TROPIC (TRansformations On Petri nets In Color) [7], [8], [9]. TROPIC has been developed in the course of the ModelCVS project [10] aiming at reusable horizontal model transformations between modeling tools supporting different languages. By employing TROPIC for debugging, one gains several advantages. Firstly, transformation designers are enabled to debug on a high-level of abstraction as TROPIC is based on a variant of CPNs, which also serve as an explicit runtime model. This also allows to employ MDA standards for debugging such as OCL [11] in order to define breakpoints and to explore details of the execution state by using queries on

* This work has been funded by the Austrian Science Fund (FWF) under grant P21374-N13.

the runtime model. Secondly, the fact that TROPIC is based on CPN concepts allows to represent QVT Relations in terms of transitions which make the afore hidden operational semantics explicit, thus allowing a white-box view on the execution of model transformations. Finally, metamodels, models, transformation logic, and trace information are included in a single homogenous debugging view thus providing the complete picture of the transformation.

The remainder of this paper is structured as follows. Section II introduces the basics of QVT Relations and TROPIC by example. In Section III, we show the translation between the concepts of QVT Relations and TROPIC which is the basis for our debugging approach. Section IV introduces an interactive debugging environment offering several features for model-based debugging of transformations. Related work is discussed in Section V, and finally, Section VI provides an outlook on future work.

II. QVT Relations and TROPIC at a Glance

To introduce the main concepts of QVT Relations and TROPIC we briefly discuss how transformation logic is specified in these two languages and illustrate the main differences, focusing on the execution of model transformations. Thereby we show how the explicit runtime model of TROPIC can be used to represent QVT Relations code as a pre-requisite for debugging on bases of an extract of the UML2Relational example [2].

A. Specification and Execution of Model Transformations

QVT Relations. By using the declarative QVT Relations language, transformation logic between two different metamodels is specified as a set of relations that must hold for the transformation to be successful. Relations contain a set of patterns used to match for existing source model elements in order to instantiate new target model elements or to modify existing ones. Since declarative approaches like QVT Relations allow for the specification of *what* has to be computed but not necessarily *how* [12], the operational semantics remains hidden. Although the QVT standard claims that the operational semantics is specified by a mapping to the low-level Core language [13], which is then executed by a transformation engine, in fact only a black-box view of the execution is provided to the transformation designer as shown in the left part of Fig. 1. The only source of information in order to verify the transformation result is trace information, indicating which source element has been transformed to which target element.

TROPIC. TROPIC uses Colored Petri Net [14] concepts, being mainly *places*, *tokens* and *transitions*, for the specification and execution of model transformations. In particular, places are derived from elements of metamodels, tokens from elements of models and transitions from the actual transformation logic (shown in the right part of Fig. 1), which are integrated in a homogenous view. The existence

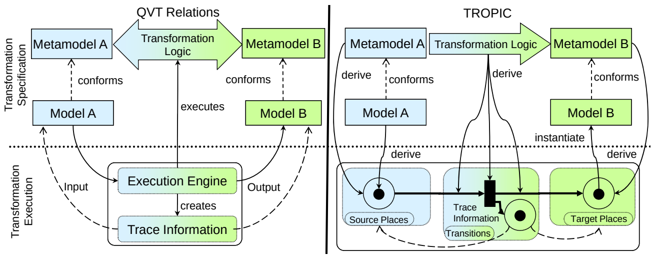


Figure 1. QVT Relations versus TROPIC

of certain model elements allows transitions to fire and thus stream tokens to the target places representing instances of the target metamodel to be created and thereby establishing trace information in terms of tokens in additional places. TROPIC, thus, provides a white-box view on model transformation execution, i.e., the specification does not need to be translated into some low-level executable artifact, but can be executed right away. Therefore, no impedance mismatch between specification and execution occurs, allowing for enhanced debuggability of model transformations.

Supported model transformation scenarios. Since transformations written in the QVT Relations language consist of declarative relations between metamodels *unidirectional* as well as *bidirectional* transformations are supported, although the actual execution requires to specify a direction. Moreover QVT Relations supports *check* and *enforce* semantics, differing in if required changes on the target side are just reported or actually undertaken, thereby supporting incremental updates which can theoretically be specified on rule level. The semantics of check and enforce, especially in combination with bidirectional model transformations is not clearly defined as stated in [15]. Furthermore, the QVT standard defines the operational semantics of QVT Relations twofold, firstly in natural language and secondly by a translation to QVT Core, being incompatible to each other [16]. This situation led to different implementations of the operational semantics in different tools, e.g. concerning the realization of *when* and *where* clauses. To circumvent these deficits of the QVT standard, the example as well as the translation presented in the following are based on the operational semantics of the mediniQVT (http://projects.ikv.de/qvt) implementation.

Although TROPIC is able to deal with incremental updates as well (cf. [17] for details) we focus on the typical model transformation scenario which creates a new target model out of an existing source model in this paper. In case of bidirectional specifications we derive two different nets, one for every execution direction. Thereby we assume that all relations specify a *checkonly* semantics for source model elements an *enforce* semantics for target model elements.

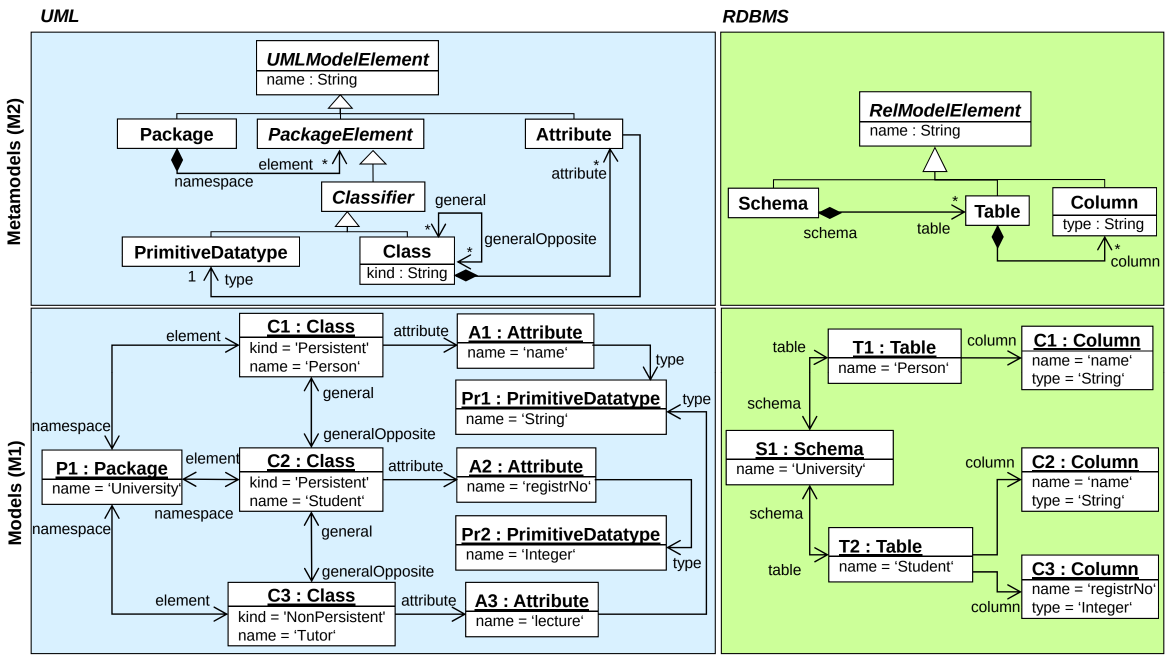


Figure 2. UML2Relational: Metamodels and Models.

B. QVT Relations to TROPIC by Example

To illustrate how TROPIC can be employed for debugging we present in the following a small part of the UML2Relational example (cf. Fig. 2), exemplifying the translation of QVT Relations to TROPIC.

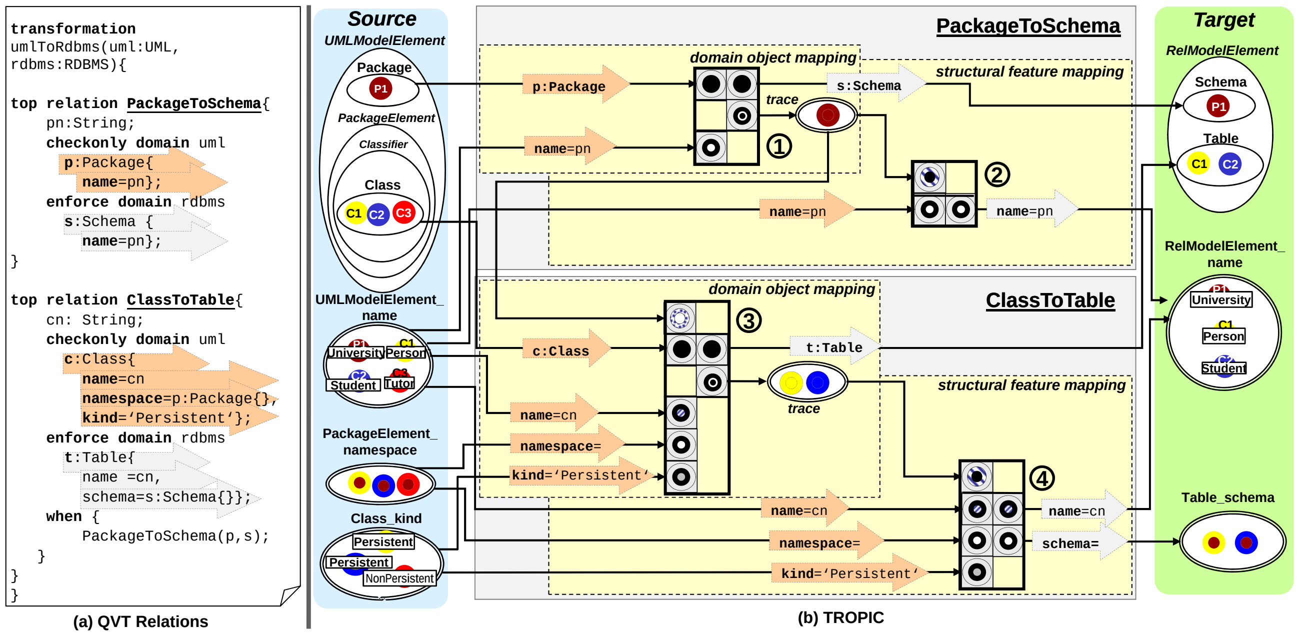
In the course of the UML2Relational example, naturally, `Packages` of UML shall be transformed into relational `Schemata` in a one-to-one fashion, whereas only persistent `Classes` shall be transformed into `Tables`. The inheritance hierarchy should be flattened, i.e., `Attributes` of base classes should be transformed to `Columns` of the table representing the inherited class. The assumed corresponding source model comprises the classes `Person`, `Student` (extending `Person`) and `Tutor` (extending `Student`) contained in the package `University`. The target model which shall be generated should contain the tables `Person` and `Student` aggregating the columns created on basis of the attributes of the base class as only these two classes were marked as persistent.

Fig. 3(a) depicts the specification of the transformation in QVT Relations, consisting of two relations for establishing the one-to-one correspondences between the two metamodels `UML` and `RDBMS`. The relations `PackageToSchema` and `ClassToTable` use an `uml` model as source model (`checkonly` semantics) and a `rdbms` model as target model (`enforce` semantics). The `PackageToSchema` relation matches for packages and their names and produces equivalent schemata and names thereof. The relation `ClassToTable` matches for persistent classes contained in a package as well as their names and creates a table labeled with the class name. The reference to the according schema is set by calling the `PackageToSchema` relation in the `when-clause` of the `ClassToTable` relation.

Even this simple example raises questions concerning specification and execution of the transformation, e.g., are there metamodel elements that will not be transformed, what happens if there are no persistent classes in a schema or in which order are the relations actually executed and the model elements created? To answer these questions, first of all, the QVT Relations code is translated to TROPIC as depicted in Fig. 3(b).

The elements of the involved metamodels are represented as places and model elements in terms of tokens residing in the corresponding places. As depicted in Fig. 5 abstract and concrete classes are both represented as *OneColoredPlaces*. Although, abstract classes cannot have instances, places created from abstract classes normally contain tokens indirectly due to other places stemming from sub-classes, being contained within them. Furthermore, the name of the class becomes the name of the place. Subclass relationships are represented by *nestedPlaces* whereby the place corresponding to the subclass is contained within the place corresponding to the superclass. The tokens contained in the “sub-place” are also visible in the “super-place”, which means that if a token is contained in a “sub-place” it may also act as input token for a transition connected to the “super-place”.

For every object, i.e., instance of `Class` that occurs in a model a *OneColoredToken* is produced, which is put into a place that corresponds to the respective element in the source



```
transformation
umlToRdbms(uml:UML,
rdbms:RDBMS){

top relation PackageToSchema{
    pn:String;
    checkonly domain uml
      p:Package{
        name=pn};
    enforce domain rdbms
      s:Schema {
        name=pn};
}

top relation ClassToTable{
    cn: String;
    checkonly domain uml
      c:Class{
        name=cn
        namespace=p:Package{},
        kind='Persistent'};
    enforce domain rdbms
      t:Table{
        name =cn,
        schema=s:Schema{}};
    when {
        PackageToSchema(p,s);
    }
}
}
```

(a) QVT Relations

(b) TROPIC

Figure 3. UML2Relational: QVT code and corresponding TROPIC specification.

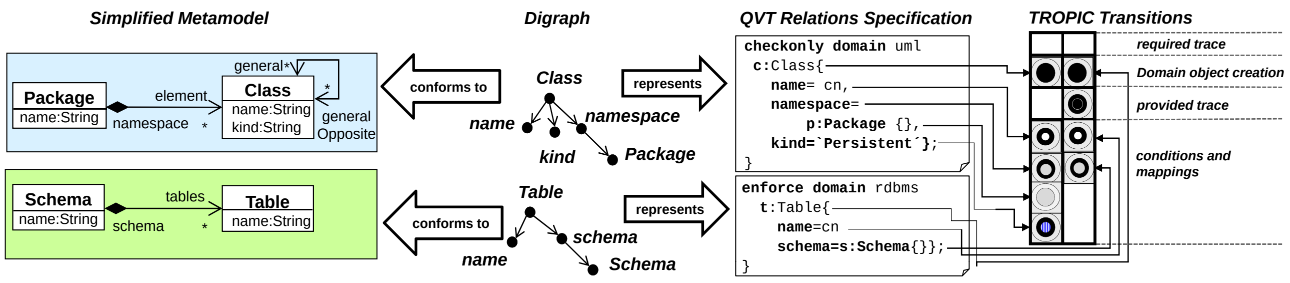
metamodel, e.g., in Fig. 3(b) a OneColoredToken representing the model element `P1` resides in the OneColoredPlace `Package`. The `color` is realized through a unique value that is derived from the object id (OID).

Attributes and references are represented by *TwoColoredPlaces*, whereby the name of the place consists of the name of the containing class and the name of the attribute or reference itself, separated by an underscore (`ClassName_name`). Notationally, the borders of two-colored places are doubly-lined to indicate that they contain two-colored tokens. Data values and links are represented by *TwoColoredTokens* whereby the *fromColor* refers to the owning element and the *toColor* represents the primitive data value or the linked target element. For the example, the link between `Class C1` and `Package P1` is represented by a TwoColoredToken residing in the TwoColoredPlace `PackageElement_namespace`.

The chosen representation of models by TROPIC let references as well as attributes become first-class citizens, resulting in a fine-grained decomposition of models. The resulting representation in combination with weak typing (all concepts depend on the Ecore types `EClass`, `EAttribute` and `EReference`, only) turned out to be especially favorable for the resolution of structural heterogeneities, a main goal of TROPIC. This is since on the one hand there are no restrictions on the order of certain transformations, like a class must be instantiated before an owned attribute and on the other hand also the special, but frequently occurring, case of schematic heterogeneities [18] can be easily dealt with, e.g., an attribute in the source model is transformed into a class in the target model by just moving the token to the respective place. At the same time, this fine grained resolution leads to numerous places, which might hinder readability but enables a detailed debugging view as transformation designer can exactly determine which (fine-grained) model elements are involved when firing a transition.

The transformation logic is represented by transitions expressing the so called *DomainPatterns* of QVT Relations which match for source elements and create target elements. DomainPatterns build digraphs conforming to the used metamodel (cf. Fig. 4 depicting the DomainPatterns of the relation `ClassToTable`), expressing correspondences between source and target metamodel elements. Unfortunately, this correspondence is hard to grasp in textual syntax. To get a visual clue which source element is transformed to which target element, TROPIC represents the nodes of such a digraph graphically (cf. Fig. 4) whereby every node is connected to a certain source or target place and nodes expressing correspondence are collocated.

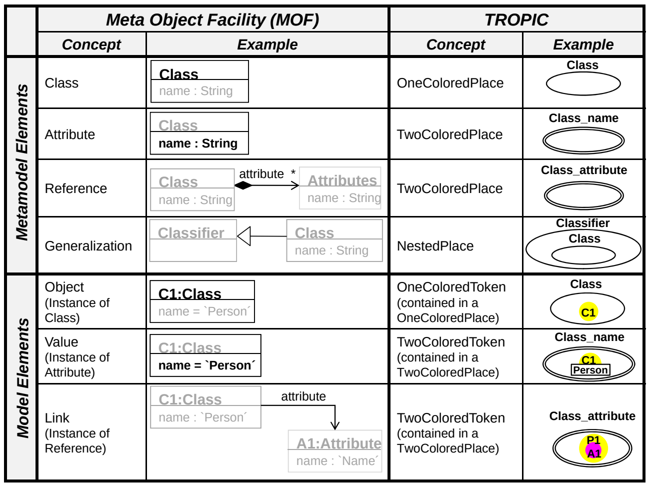
Transitions consist of input placeholders (LHS of the transition) representing the pre-conditions of a certain transformation, whereas output placeholders (RHS of the transition) depict its post-condition. To express these pre- and post-conditions, so-called meta tokens are used, prescribing a certain token configuration by means of colors. By matching a certain token configuration from the input places, i.e., fulfilling the pre-condition, the transition is ready to fire, with the colors of the input tokens being bound to the meta tokens residing in the input placements. The production of output tokens fulfilling the post-condition once a transition fires is dependent on the matched input tokens. For example, when a simple one-to-one correspondence should be



```
checkonly domain uml
 c:Class{
   name= cn,
   namespace=
       p:Package {},
   kind=`Persistent´};
}
```

```
enforce domain rdbms
  t:Table{
   name=cn
   schema=s:Schema{}};
}
```

Figure 4. Dependencies between Metamodels, QVT, and TROPIC.



	Meta Object Facility (MOF)		TROPIC	
	Concept	Example	Concept	Example
Metamodel Elements	Class	Class / name : String	OneColoredPlace	Class
	Attribute	Class / name : String	TwoColoredPlace	Class_name
	Reference	Class / name : String; attribute *; Attributes / name : String	TwoColoredPlace	Class_attribute
	Generalization	Classifier; Class / name : String	NestedPlace	Classifier; Class
Model Elements	Object (Instance of Class)	C1:Class / name = `Person´	OneColoredToken (contained in a OneColoredPlace)	Class; C1
	Value (Instance of Attribute)	C1:Class / name = `Person´	TwoColoredToken (contained in a TwoColoredPlace)	Class_name; C1; Person
	Link (Instance of Reference)	C1:Class / name : `Person´; attribute; A1:Attribute / name : `Name´	TwoColoredToken (contained in a TwoColoredPlace)	Class_attribute; P1; A1

Figure 5. Representing MOF concepts within TROPIC

implemented, the colors of input and output meta tokens are equal meaning that a token is streamed through the transition only. Starting from the domain object (i.e., the digraph's root node of Fig. 4) which is represented in TROPIC by a one colored meta token within transitions, navigation in the graph is enabled using two colored meta tokens whereby the outer color represents the source object and the inner color represents the target object of the link (e.g., the `namespace` link in Fig. 4) or the object and its primitive value in case of attributes respectively (e.g., the `name` attribute in Fig. 4).

The graphical representation in TROPIC offers the possibility to debug QVT Relations code on a high level of abstraction [19]. Interactive debugging is enabled by stepwise firing of transitions. For example in the above scenario first a `Schema` is created and then `Tables` are added since transition (3) in Fig. 3(b) is only enabled if a `Schema` has already been created which is determined by the trace information of transition (1). This shows that with the given QVT specification a `Schema` is created irrespective if it contains persistent classes or not. Please note that model transformations demand for a specific consumption behavior. If transitions (3) in Fig. 3 would consume (which is the default in CPNs) the trace token `P1`, transition (3) could only fire once and the 1:n relationship between package and classes would not be correctly transformed to schemas and tables. Therefore, TROPIC does not consume the tokens per default, but only store the combination of tokens currently fulfilling the precondition in a so called *execution history*. This allows transitions to fire for all possible combinations, which is typically desired in transformation scenarios. The execution history additionally enables an undo/redo mechanism and thus allows transformation designers to easily step forwards and backwards in the transformation process. Note that the tokens in the target places result from successfully executed transitions meaning that Fig. 3 shows the final state of a model transformation.

III. Translating QVT Relations to TROPIC

The previous section introduced the translation of QVT Relations to TROPIC at a glance. Now, the translation is described in detail based on the metamodels of both transformation languages (cf. Fig. 6).

Representation of source and target metamodels and models. QVT Relations as well as TROPIC provide containers (cf. *metaclasses RelationTransformation* and *Net*, respectively) for aggregating metamodels, models and transformation logic. Whereas QVT Relations simply represents the involved metamodels as a whole (cf. metaclass *TypedModel*), e.g., `uml` and `rdbms` in the previous example, TROPIC explicitly represents each element of the metamodels as first class concepts in terms of a place (c.f. metaclass *Place*).

Regarding models, QVT provides no explicit representation mechanism, which is again in contrast to TROPIC, where each model element is explicitly represented by *OneColoredTokens* residing in corresponding *OneColoredPlaces* in case of classes and objects or by *TwoColoredTokens* in residing in corresponding *TwoColoredPlaces* in case of attributes and values or references and links.

Representation of transformation logic. To aggregate transformation logic, both, QVT Relations and TROPIC provide a container (cf. metaclasses *Relation* and *TropicUnit*, respectively). To incorporate the involved metamodels, QVT Relations uses *RelationDomains* which bind *Relations* to the source or target metamodel. In TROPIC, *Arcs* connect the metamodel elements (places) with the transformation

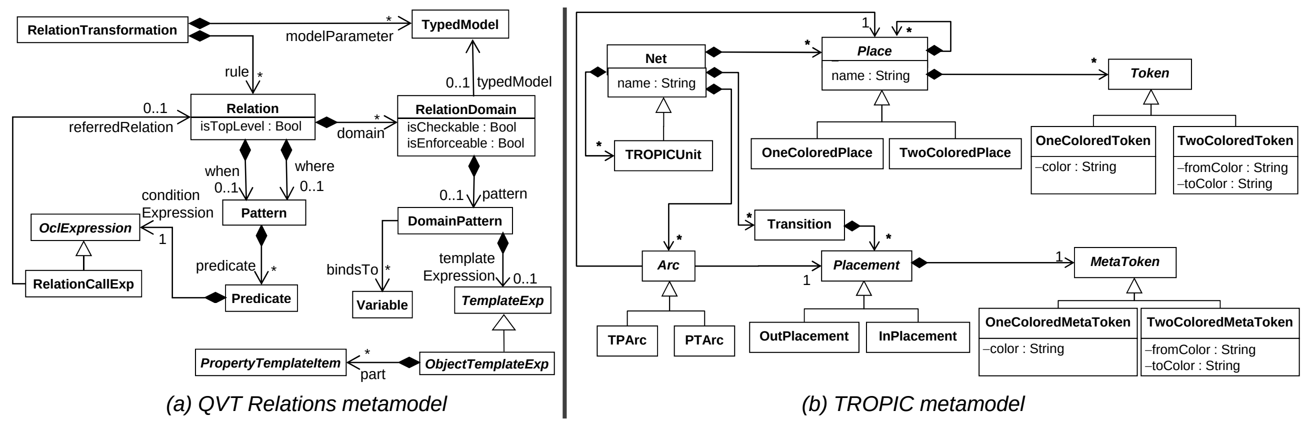


Figure 6. Metamodels of (a) QVT Relations and (b) TROPIC.

logic (transitions) whereby the actual direction of an arc is derived from the user-defined execution direction of the QVT Relations.

In QVT, *DomainPatterns* specify the selection of model elements forming, as mentioned before, a digraph conforming to a metamodel using the specified domain object as root node (cf. Fig. 4). The graph consists of objects (cf. class *ObjectTemplExp*), attributes and links—both represented by the class *PropertyTemplateItem*. In contrast, TROPIC fires *Transitions* defining their behavior in terms of *Placements* which contain *MetaTokens* either matching for objects (class *OneColoredMetaToken*) or attribute values and links (class *TwoColoredMetaToken*). The relation `PackageToSchema` in Fig. 3 shows that every part of the DomainPattern of the source domain, being `p:Package` and `name=pn` is represented as a corresponding *InPlacement* in the transitions whereby `p:Package` is represented as a OneColoredMetaToken and `name=pn` is represented as a TwoColoredMetaToken. The DomainPattern of the target domain is represented by according *OutPlacements* whereby same colored *MetaTokens* are used to express correspondences between the source and the target domain.

QVT Relations allows to specify *DomainPatterns* containing references of the target model having a multiplicity greater than one, e.g., a `Schema` can contain an arbitrary number of `Tables`. To ensure that the target domain object (e.g., a `Schema`) is created only once, a QVT Relations transformation engine examines the trace information. In TROPIC, we therefore separate the creation of the domain object from its containing features, whereby the transition transforming the domain object produces trace information used by the transitions mapping the contained features (cf. transitions (1) and (2) in Fig. 3(b)). Trace information in TROPIC is furthermore used to express *when-* and *where-clauses* which is the QVT Relations concept to call dependent relations.

Table I summarizes the described mappings between QVT Relations and TROPIC concepts.

IV. TROPIC Debugging Environment

The graphical representation of QVT Relations code in TROPIC as described in the previous section, is advantageous with respect to an explicit and homogenous representation of all artifacts of a transformation, but is in fact, just a first step towards a model transformation debugging environment. Beyond that, the TROPIC debugging environment provides mechansims in order to support the three main phases of debugging [20], i.e., observing facts, tracking origins, and fixing bugs, which is described in the following, first at a glance and then in detail by example.

A. Debugging Environment at a Glance

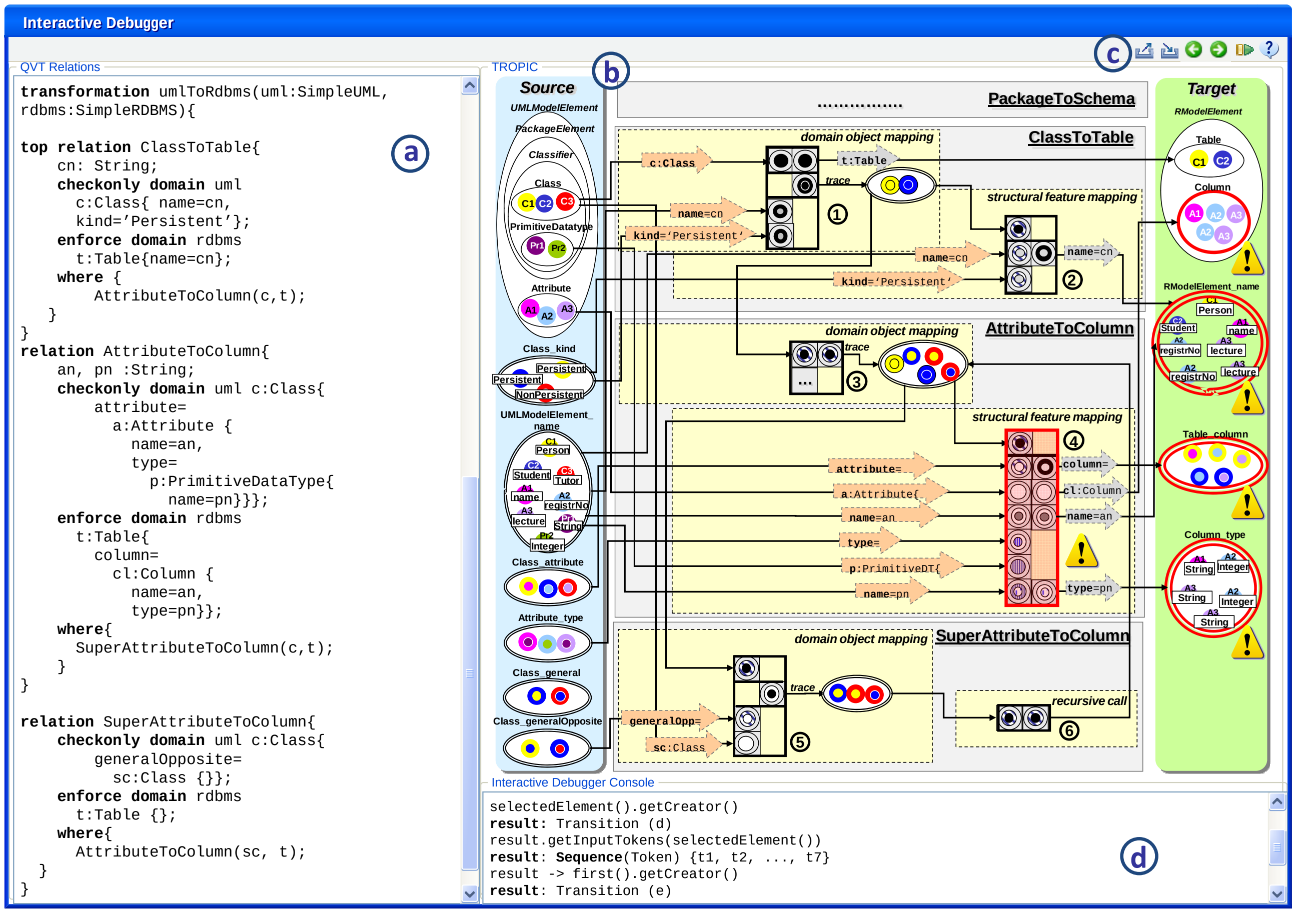
The TROPIC debugging environment is based on Eclipse and includes two editors, one presenting the QVT Relations in textual syntax (cf. Fig. 7(a)) and another one that shows the graphical representation thereof in TROPIC (cf. Fig. 7(b)). We assume a syntactically correct QVT Relations specification since only in this case we can guarantee a correct translation to TROPIC and the propagation of changes in TROPIC back to QVT Relations. The TROPIC editor toolbar (cf. Fig. 7(c)) provides common debugging functionalities such as enabling stepwise debugging to figure out the operational semantics by firing transitions including an undo/redo mechanism. Furthermore, functionalities are provided to save the generated target model, i.e., to switch from the token representation to a model representation, and to load a new source model into the debugging environment. For testing purposes, an expected target model can be loaded, which is automatically compared to the target model actually created by the transformation.

Besides these standard debugging functionalities, there are additional debugging features resulting as a benefit of using a dedicated runtime model. In particular, OCL is employed for two different debugging purposes. First, OCL is used to define conditional breakpoints at different levels of

Table I
QVT RELATIONS TO TROPIC TRANSLATION

QVT Relation Concept	TROPIC Concept
RelationTransformation	Net
TypedModel	One-/TwoColoredPlace
n.a. (Model element)	Token
Relation	TropicUnit
Execution direction	Arc
DomainPattern	Transition
ObjectTemplateExp	Placement + OneColoredMetaToken
PropertyTemplateItem	Placement + TwoColoredMetaToken
Variable	Color of MetaToken
When-clause	InPlacement + PTArc from dependent trace place
Where-clause	Trace place + TPArc to InPlacement

granularity. Thus, it can not only be defined that execution should stop if, e.g. a certain token is streamed into a certain place, but also if tokens occur in several different places. Second, OCL is used to tackle the well-known problem of debugging environments that programs execute forward in time whereas programmers must reason backwards in time to find the origin of a bug. For this, a dedicated debugging console based on the *Interactive OCL Console* of Eclipse (cf. Fig. 7(d)) is supported, providing several pre-defined debugging functions to explore and to understand the history of a transformation by determining and tracking paths of produced tokens.



```
transformation umlToRdbms(uml:SimpleUML,
rdbms:SimpleRDBMS){

top relation ClassToTable{
    cn: String;
    checkonly domain uml
      c:Class{ name=cn,
      kind='Persistent'};
    enforce domain rdbms
      t:Table{name=cn};
    where {
        AttributeToColumn(c,t);
   }
}
relation AttributeToColumn{
    an, pn :String;
    checkonly domain uml c:Class{
        attribute=
          a:Attribute {
            name=an,
            type=
              p:PrimitiveDataType{
                name=pn}}};
    enforce domain rdbms
      t:Table{
        column=
          cl:Column {
            name=an,
            type=pn}};
    where{
      SuperAttributeToColumn(c,t);
    }
}

relation SuperAttributeToColumn{
    checkonly domain uml c:Class{
        generalOpposite=
          sc:Class {}};
    enforce domain rdbms
      t:Table {};
    where{
      AttributeToColumn(sc, t);
  }
}
```

```
selectedElement().getCreator()
result: Transition (d)
result.getInputTokens(selectedElement())
result: Sequence(Token) {t1, t2, ..., t7}
result -> first().getCreator()
result: Transition (e)
```

Figure 7. Debugging Environment showing the transformation of Attributes to Columns

Table II
OCL OPERATIONS FOR DEBUGGING

Context	QCL Operation	Description

Place	`getMatchingTokens:Set(Token)`
	`getMismatchedTokens:Set(Token)`
		tokens that match a transition
		tokens not matching a transition

Token	`getCreator:Transition`	transition that created a token

Transition	`getInputTokens(Token):Set(Token)`
	`whyNotActivated:Set(InPlacement)`
		source tokens of a transition
		InPlacements that fail

InPlacement	`getMatchingTokens:Set(Token)`	tokens that match a condition

B. Debugging Environment By-Example

Fig. 7 shows the QVT Relations code and the corresponding TROPIC specification for transforming `Attributes` to `Columns` to complete our transformation example of Section II-B. As mentioned before, in addition to the simple one-to-one correspondence between `Attributes` and `Columns` (cf. relation AttributeToColumn), the inheritance hierarchy should be flattened, i.e., `Attributes` of base classes should be transformed to `Columns` of the table of the inherited class (cf. relation SuperAttributeToColumn). For demonstration purposes, the transformation specification, however, contains a bug. In the following, it is described how this bug is observed, tracked, and fixed using the TROPIC debugging environment.

Observing Facts. Observation determines facts about what has happened in a concrete run of a transformation. The first possibility of observing facts within our environment is either to simulate the transformation and watch for unexpected behavior or to debug the transformation step-by-step. In order to detect unexpected behavior automatically, the resulting target model can be compared to an expected target model to identify wrong or missing target tokens. If such faulty parts of the target model are detected, the owning target places as well as the transitions that produce tokens in these places are highlighted to ease finding the reasons for the errors. Comparing the resulting elements of the UML2Relational example, i.e., the tokens in the target places in Fig. 7, to the expected ones depicted in the bottom right part of Fig. 2, one can easily recognize that too many columns have been created. Therefore, the place `Column` as well as transition (4) targeting this place are highlighted. Due to the incorrect amount of columns, all places representing properties of a column contain errors too and are therefore highlighted as well. By examining the tokens in the place `Column`, one can see that it contains tokens of the non-persistent class `Tutor` (labeled with `A3`). However, it is not obvious why these additional tokens are created as there are possibly many tokens and transitions involved in their creation.

Tracking Origins. Once an error has been observed during debugging, the origin has to be discovered by reasoning backwards in time, questioning e.g, why the tokens labeled with `A3` in the place `Column` have been created. The graphical representation shows that the tokens have been created by transition (4) but we do not know which source tokens were used to create exactly these target tokens. To get this information, the transformation designer may use the interactive debugging console. Within this console, s/he can use standard OCL functions and pre-defined OCL debugging functions to formulate queries that can be invoked on the runtime model, i.e., on the TROPIC representation.

In our example, the transformation designer selects the bottom right token labeled with `A3` of the place `Column`, representing a wrongly created column `lecture` and invokes the function `getCreator` (cf. line 1 in the debugging console) which highlights the transition (4) in the editor and additionally returns the result of the function in the debugging console. This transition receives an input token from a trace place (represented by the topmost LHS metatoken) which is filled by two transitions, namely by transition (3) and (6). Therefore, it is not clear which one of these two transitions is responsible for providing the trace token used for creating the selected `Column` token. To determine the responsible transition, the developer invokes the function `getInputTokens(selectedElement)` on transition (4) (cf. line 2 in the debugging console) returning a sequence of input tokens which has been bound to produce the `selectedElement`. The elements in this sequence are ordered regarding to their graphical location within the transition, thus the first token in the sequence is the token that matched the topmost LHS metatoken. To get this token, the transformation designer applies the standard OCL function `first()` on the previously computed sequence which returns the single trace token. Now, the transformation designer again applies the function `getCreator()` to determine which transition is responsible for producing this trace token, which is transition (5), as transition (6) only streams the token through. Taking a closer look on transitions (5), one can see that this transition uses tokens of the wrong source place, namely `Class_generalOpposite` instead of `Class_general` (see error sign in Figure 7), being the origin of the error. The respective QVT domain pattern selects the wrong class which is then handed to the dependent `AttributeToColumn` relation adding the

Table III
POSSIBLE OPERATIONS FOR BUG FIXING

QVT Concept	TROPIC Concept	Add	Del	Edit
RelationTransformation	Net	x	x	x
TypedModel	Place	x	x	x
n.a.	Token	✓	✓	✓
Relation	TropicUnit	∼	∼	✓
RelationDomain	Arc	∼	∼	x
DomainPattern	Transition	∼	∼	✓
TemplateExp/PropertyTemplateItem	Placement/MetaToken	✓	✓	✓

✓ applicable and recommended, ∼ applicable but not recommended x not applicable

wrong columns to a table. This error results in the fact that, e.g., for the class `Student` the derived class `Tutor` is selected instead of the base class `Person` and therefore a column `Lecture` is created. The currently available predefined OCL operations on TROPIC elements are outlined in Table II.

Fixing Bugs. After finding the origins of a bug, it is possible to adapt the transformation logic during debugging directly in TROPIC and propagate the changes back to QVT Relations. We mainly focus on bugs in *DomainPatterns*, as fixing those bugs reflects minor changes in QVT Relations and thus can be updated in the debugging environment during the debugging process.

As described in Section II-B and depicted in Fig. 4, the graph of the domain pattern corresponds to a metamodel and is represented in TROPIC in terms of *Placements* aggregated in *Transitions*. When fixing a bug in TROPIC we have to ensure that the graph remains valid. For example, if a new precondition should be added to a transition it must be ensured that the new *InPlacement* connects to a place which represents a possible new leave in the graph. Assuming the class `Person` as domain object, it would be possible to add an *InPlacement* representing the link `attribute` but it is not possible to ask for the `name` property of an attribute without adding the link, since `name` is no possible leave. To hide this complexity from the transformation designer, every transition is aware of its domain object and the graph in terms of the already contained *Placements*. Thus, it is possible to calculate a list of possible new leaves which is presented to the transformation designer during bug fixing. By choosing an entry of the list, the corresponding *Placement*, the *MetaToken* as well as the *Arc* to the corresponding place are created automatically and added to the transition in TROPIC and as additional condition in the QVT Relations specification. If an element of a domain pattern is deleted, all descendent child nodes in the graph are deleted as well. Please note that we do not allow the deletion of the domain object as we would otherwise loose the possibility to calculate the graph. If such a fundamental modification is necessary, the domain must be changed in the QVT Relations specifications and debugging must be restarted. Furthermore during debugging it is impossible to change the metamodels of the transformation as this would result in serious changes in the transformation logic. Although model elements are not represented in QVT Relations, we see that it is also possible to add, delete, and edit tokens, which is especially useful to alter the source models during debugging, e.g., by specifying missing attributes. Summarizing, Table III shows which bug fixing actions are allowed on which QVT concept.

In our example too many `Columns` have been created, because of the wrong specified reference. To collect the attributes of the base classes, we first delete the wrong `generalOpposite` restriction of transition (e) and then add the correct `general` restriction which actually represents the link to the parent class by selecting the appropriate entries in the presented list. In the QVT Relations specification therefore the restriction `generalOpposite=sc:Class{}` is updated to `general=sc:Class{}` in the relation `SuperAttributeToColumn`.

V. RELATED WORK

The main objective of this paper is to enhance the debuggability of QVT Relations by translating them into a CPN based formalism. Therefore, we consider five orthogonal threads of related work. First, we point out the debugging support of existing QVT Relations environments. As we transform QVT Relations to Colored Petri nets we secondly examine other translational approaches and thirdly take a look if there are other approaches available using Petri nets for model transformation. After comparing debugging support of other transformation languages to our approach we finally present related work concerning methods for debugging model transformations in general.

QVT Relation environments. As mentioned by Kurtev et al. [4], tool support for QVT is still in its infancy. The most advanced tool seems to be mediniQVT[1] which also provides some basic debugging features. These features are, however, fully based on the Eclipse debugger which allows transformation designers to inspect variables in a certain execution state only. This makes it hard to recognize what is really going on during a transformation, because neither

[1]http://projects.ikv.de/qvt

the output model nor the trace information can be accessed before the transformation has been finished. Other QVT Relations tools such as ModelMorf[2] or MOMENTQVT[3] do not provide debugging facilities.

Translational approaches for QVT Relations. Besides dedicated QVT Relations environments, so-called translational approaches have been proposed for executing QVT Relations on top of existing technologies. Jouault and Kurtev [13] propose to execute QVT Relations within the ATL Virtual Machine (ATL VM), by transforming QVT Relations into ATL VM code. Romeikat et al. [21] transform QVT Relations into the QVT Operational Mappings language and execute the result with tools such as SmartQVT[4]. These two approaches transform QVT Relations into code on a lower level of abstraction and seem to be therefore not suitable for debugging QVT Relations adequately. Greenyer and Kindler [22] propose to transform QVT Relations into Triple Graph Grammars (TGGs) which can be executed in TGGs tools such as Fujaba[5]. Because QVT Relations and TGGs are conceptually and also syntactically similar, one can remain on the same abstraction level. The debugging problem is, however, shifted only, since TGGs are not directly executable within existing tools. Again, TGGs have to be translated into executable instructions which are not suitable for debugging QVT Relations.

Petri nets and model transformations. The relatedness of Petri nets and graph rewriting systems has also induced some impact in the field of model transformation. Especially in the area of graph transformations some work has been conducted that uses Petri nets to check formal properties of graph production rules. Thereby, the approach proposed in [23] translates individual graph rules into a place/transition net and checks for its termination. Another approach is described in [24], which applies a transition system for modeling the dynamic behavior of a metamodel. Compared to these two approaches, our intention to use Petri nets is fundamentally different. While these two approaches are using Petri nets as a back-end for automatically analyzing properties of transformations, we are using Petri nets as a front-end to foster debuggability as well as to explore formal properties.

Debugging Support and Understandability of Transformation Languages. In general there is hardly any debugging support for transformation languages. Most often only low-level information of the execution engine is provided, but an according traceability to the corresponding higher-level mapping specifications is missing. For example, in the Fujaba environment, a plugin called MoTE [25] compiles TGG rules [26] into Fujaba story diagrams that are implemented in Java, which obstructs a direct debugging on the level of TGG rules. In [27] the generated source code is annotated accordingly to allow the visualization of debugging information in the generated story diagrams, but not on TGG level. Additional to that, Fujaba supports visualization of how the graph evolves during transformation, and allows interactive application of transformation rules. Furthermore, approaches like VIATRA [28] producing debug reports that trace an execution, only, are likewise considered inadequate for debugging since a minimum requirement for the debugging should be the ability to debug at least whole transformation rules, by which we refer to as the stepwise execution and inspection of the execution state. The debugging of ATL [29] is based on the stepwise execution of a stack-machine that interprets ATL bytecode, which also allows observing the execution of whole transformation rules. SmartQVT and TefKat [30] allow for similar debugging functionality.

What sets TROPIC apart from these approaches is that all debugging activities are carried out on a single integrated formalism, without needing to deal with several different views. Furthermore, this approach is unique in allowing interactive execution not only by choosing rules or by manipulating the state directly, but also by allowing to modify the structure of the net itself. This ability for live-programming enables an additional benefit for debugging and development: one can correct errors (e.g., stucked tokens) in the net right away without needing to recompile and restart the debug cycle.

Concerning the understandability of model transformations in terms of a visual representation and a possibility for a graphical simulation, only graph transformation approaches like, e.g., Fujaba allow for a similar functionality. However, these approaches neither provide an integrated view on all transformation artifacts nor do they provide an integrated view on the whole transformation process in terms of the past state, i.e., which rules fired already, the current state, and the prospective future state, i.e., which rules are now enabled to fire. Therefore, these approaches only provide snapshots of the current transformation state.

Debugging model transformations. Hibberd et al. [31] present forensic debugging techniques for model transformations by utilizing the trace information of model transformation executions for determining the relationships between source elements, target elements, and the involved transformation logic. With the help of such trace information, it is possible to answer debugging questions implemented as queries which are important for localizing bugs. In addition, they present a technique based on program slicing for further narrowing the area where a bug might be located. The work of Hibberd et al. is orthogonal to our approach, because we are using live debugging techniques instead of forensic mechanisms. However, our approach allows to answer debugging questions based on the visualization of the path a source token has taken to become a target token.

[2]http://www.tcs-trddc.com/ModelMorf
[3]http://moment.dsic.upv.es/ content/view/34/75/
[4]http://smartqvt.elibel.tm.fr/
[5]http://wwwcs.upb.de/cs/fujaba

VI. Conclusion and Further Work

In this paper we proposed a graphical debugger for QVT Relations based on TROPIC. By accomplishing debugging in TROPIC, one gains several advantages, being, firstly, the high level of abstraction, secondly, the explicit operational semantics and thirdly, the homogenous representation of all transformation artifacts. Furthermore we showed a debugging environment helping programmers to observe facts, find the origins of errors and correct them in the debugger itself.

Several issues for future work remain open. As already mentioned the QVT standard defines the operational semantics of QVT Relations twofold, leading to the situation there exist different implementations of the operational semantics in different tools. Currently, our translation is based on the implementation of mediniQVT, but we are planning to investigate the implementations of different tools. One part of QVT Relations, that has been neglected so far is the integration of queries in the TROPIC representation. Furthermore, as TROPIC is based on a variant of CPNs we will explore if Petri Net properties can be used to check for potential shortcomings in QVT Relations specifications. Additionally, the current visualization of the transformation logic by means of transitions containing a set of InPlacements and OutPlacements can easily become hard to comprehend when domain patterns grow larger. Therefore, other visualization techniques should be employed, e.g., by embedding object diagrams for describing the pre- and post-conditions of transitions.

References

[1] Object Management Group, "MDA Guide Version 1.0.1," http://www.omg.org/docs/omg/03-06-01.pdf, 2003.

[2] ——, "Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification," www.omg.org/docs/ptc/07-07-07.pdf, 2007.

[3] P. Wadler, "Why no one uses functional languages," *SIGPLAN Not.*, vol. 33, no. 8, pp. 23–27, 1998.

[4] I. Kurtev, "State of the Art of QVT: A Model Transformation Language Standard," in *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany*. Springer, 2007, pp. 377–393.

[5] P. Stevens, "A Landscape of Bidirectional Model Transformations," in *Generative and Transformational Techniques in Software Engineering II: Int. Summer School*. Braga, Portugal: Springer LNCS 5235, 2007, pp. 408–424.

[6] M. Wimmer, A. Kusel, J. Schoenboeck, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reviving QVT Relations: Model-based Debugging using Colored Petri Nets," in *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09), to be published*, Denver, USA, 2009.

[7] T. Reiter, M. Wimmer, and H. Kargl, "Towards a runtime model based on colored Petri-nets for the execution of model transformations," in *Proceedings of 3rd Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD*, Berlin, Germany, 2007, pp. 19–23.

[8] G. Kappel, H. Kargl, T. Reiter, W. Retschitzegger, W. Schwinger, M. Strommer, and M. Wimmer, "A Framework for Building Mapping Operators Resolving Structural Heterogeneities," in *Proceedings of Information Systems and e-Business Technologies (UNISCON'2008)*. Klagenfurth, Austria: Springer LNBIP 5, 2008, pp. 158–174.

[9] M. Wimmer, A. Kusel, T. Reiter, W. Retschitzegger, W. Schwinger, and G. Kappel, "Lost in Translation? Transformation Nets to the Rescue!" in *Proceedings of the 8th Int. Conf. on Information Systems Technology and its Applications*. Sydney, Australia: Springer LNBIP 20, 2009, pp. 315–327.

[10] M. Wimmer, T. Reiter, H. Kargl, G. Kramler, E. Kapsammer, W. Retschitzegger, W. Schwinger, and G. Kappel, "Lifting metamodels to ontologies - a step to the semantic integration of modeling languages," in *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MODELS'06)*. Genua, Italy: Springer LNCS 4199, 2006, pp. 528–542.

[11] Object Management Group, "OCL Specification Version 2.0," http://www.omg.org/docs/ptc/05-06-06.pdf, 2005.

[12] J. W. Lloyd, "Practical Advantages of Declarative Programming," in *Proceedings of the Joint Conference on Declarative Programming (GULPRODE'94)*, Peñiscola, Spain, 1994, pp. 18–30.

[13] F. Jouault and I. Kurtev, "On the architectural alignment of ATL and QVT," in *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*. Dijon, France: ACM, 2006, pp. 1188–1195.

[14] K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modeling and Validation of Concurrent Systems*. Springer, 2009.

[15] P. Stevens, "Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions," in *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS'07)*. Nashville, USA: Springer LNCS 4735, 2007, pp. 1–15.

[16] ——, "A simple game-theoretic approach to checkonly QVT Relations," in *Proceedings of ICMT2009 - International Conference on Model Transformation Theory and Practice of Model Transformations*. Zurich, Switzerland: Springer LNCS 5563, 2009, pp. 165–180.

[17] T. Reiter, "T.r.o.p.i.c.: Transfromations on petri nets in color," Ph.D. dissertation, Johannes Kepler University Linz, Faculty of Bioinformatics, Februar 2008.

[18] F. Legler and F. Naumann, "A Classification of Schema Mappings and Analysis of Mapping Tools," *Proceedings of Datenbanksysteme in Business, Technologie und Web (BTW 2007), Germany, Aachen*, 2007.

[19] A. Kusel, W. Schwinger, M. Wimmer, and W. Retschitzegger, “Common Pitfalls of Using QVT Relations - Graphical Debugging as Remedy,” in *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009)*. Potsdam, Germany: IEEE Computer Society, 2009, pp. 329–334.

[20] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.

[21] R. Romeikat, S. Roser, P. Müllender, and B. Bauer, “Translation of QVT Relations into QVT Operational Mappings,” in *Proceedings of 1st International Conference on Theory and Practice of Model Transformations*. Zurich, Switzerland: Springer LNCS 5063, 2008, pp. 137–151.

[22] J. Greenyer and E. Kindler, “Reconciling TGGs with QVT,” in *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems (MODELS’07)*. Nashville, USA: Springer LNCS 4735, 2007, pp. 16–30.

[23] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer, “Termination Analysis of Model Transformations by Petri Nets,” in *Proceedings of 3rd International Conference on Graph Transformations*. Natal, Brazil: Springer LNCS 4961, 2006, pp. 260–274.

[24] J. de Lara and H. Vangheluwe, “Translating Model Simulators to Analysis Models,” in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE’08)*. Budapest, Hungary: Springer LNCS 4961, 2008, pp. 77–92.

[25] R. Wagner, “Developing Model Transformations with Fujaba,” in *Proceedings of the 4th International Fujaba Days 2006*, Bayreuth, Germany, 2006, pp. 79–82.

[26] A. Koenigs, “Model Transformation with Triple Graph Grammars,” in *Proceedings of Model Transformations in Practice Workshop of MODELS’05*. Montego Bay, Jamaica: Springer LNCS 3844, 2005.

[27] L. Geiger, “Model Level Debugging with Fujaba,” in *Proceedings of 6th International Fujaba Days*, Dresden, Germany, September 2008, pp. 23–28.

[28] A. Balogh and D. Varró, “Advanced model transformation language constructs in the VIATRA2 framework,” in *SAC ’06: Proceedings of the 2006 ACM Symposium on Applied Computing*. Dijon, France: ACM, April 2006, pp. 1280–1287.

[29] F. Jouault and I. Kurtev, “Transforming Models with ATL,” in *Proceedings of Model Transformations in Practice Workshop of MODELS’05*. Montego Bay, Jamaica: Springer LNCS 3844, 2005, pp. 128–138.

[30] M. Lawley and J. Steel, “Practical Declarative Model Transformation with Tefkat,” in *Proceedings of Model Transformations in Practice Workshop of MODELS’05*. Montego Bay, Jamaica: Springer LNCS 3844, 2005, pp. 139–150.

[31] M. Hibberd, M. Lawley, and K. Raymond, “Forensic Debugging of Model Transformations,” in *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS’07)*. Nashville, USA: Springer LNCS 4735, 2007, pp. 589–604.