Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Colouring: Execution, Debug and Analysis of QVT-Relations Transformations through Coloured Petri Nets

Esther Guerra [⋆], **Juan de Lara**

Universidad Autónoma de Madrid (Spain), e-mail: {Esther.Guerra, Juan.deLara}@uam.es

Received: date / Revised version: date

Abstract QVT is the standard language sponsored by the OMG to specify model-to-model transformations. It includes three different languages, being QVT-Relations (QVT-R) the one with higher-level of abstraction. Unfortunately, there is scarce tool support for it nowadays, with incompatibilities and disagreements between the few tools implementing it, and lacking support for the analysis and verification of transformations. Part of this situation is due to the fact that the standard provides only a semi-formal semantics for QVT-R.

In order to alleviate this situation, this paper provides a semantics for QVT-R through its compilation into Coloured Petri nets. The theory of Coloured Petri nets provides useful techniques to analyse transformations (e.g. detecting relation conflicts, or checking whether certain structures are generated or not in the target model) as well as to determine their confluence and termination given a starting model. Our semantics is flexible enough to permit the use of QVT-R specifications not only for transformation and check-only scenarios, but also for model matching and model comparison, not covered in the original standard.

As a proof of concept, we report on the use of CPNTools for the execution, debugging, verification and validation of transformations, and on a tool chain (named *Colouring*) to transform QVT-R specifications and their input models into the input format of CPNTools, as well as to export and visualize the transformation results back as models.

Key words Model Driven Engineering – Model-to-Model Transformations – QVT-Relations – Coloured Petri Nets – Validation and Verification

[⋆] *Present address:* Computer Science Department, Universidad Autónoma de Madrid, 28049 Madrid (Spain)

1 Introduction

Model Driven Engineering (MDE) is a software engineering paradigm that promotes an active use of models to conduct the software development process. In this way, models are not used just as a passive documentation, but to generate code, test and verify the applications under construction. Hence, models become first-class citizens, and their manipulation a key activity in MDE [41].

The purpose of Model-to-Model (M2M) transformation is translating a model from a source to a target language. This activity is one of the pillars of MDE, where it is used to refine and abstract models, for transformation into a semantic analysis domain, or for language migration. Among the existing M2M transformation languages, QVT (for Query/View/Transformation) [37] stands out for being the transformation standard proposed by the OMG in the framework of the Model Driven Architecture [32]. QVT includes three different languages and has a hybrid declarative/imperative nature. The declarative part provides a user-friendly, high-level language called *Relations* (QVT-R) whose semantics is given by its compilation into a lower-level language called *Core* (QVT-C). In its turn, the imperative part provides a language called *Operational mappings* (QVT-O).

Despite the popularity of the QVT standard, few tools support the execution of QVT-R [31,33], and even less its verification or validation. Furthermore, some authors have reported disagreements of these tools with respect to semantic issues [43]. This is partly due to the fact that the semantics of QVT-R is given in terms of QVT-C, which in its turn is semi-formally defined. In addition, the fact that several languages with different semantics are sometimes described as "QVT-like" [4,22] has also contributed to the confusion with respect to how QVT-R works. Thus, the MDE community would benefit from a semantics for QVT-R enabling the analysis of transformations, and a framework that explains and improves the understanding of how QVT-R transformations work. Our aim is to contribute in this direction.

The QVT standard defines several usage scenarios for QVT-R. In the *transformation scenario*, a QVT-R specification is used to create a target model from a source model, whereas in the *check-only scenario*, it is used to check whether a target model is consistent with a source one. Although not considered by the standard, we propose using the *same specification* for other purposes as well, for example in the context of model comparison and model matching [16,24,27]. In these scenarios, the aim is generating all possible traces between two models in order to investigate their similarities, or as a previous step towards their synchronization (updating both models to achieve consistency) or merging (producing a model that merges elements considered equivalent) [25]. While in the transformation and checking scenarios the evaluation is directional (from source to target), in the comparison scenario we should consider the source and target *at the same time* to create the trace model. However, this variation is not taken into account in the standard, and is not supported by existing tools.

Coloured Petri nets (CP-nets or CPNs in short) [18, 19] is a formalism for modelling, simulation and analysis of systems in which concurrency, communication and synchronization are salient features. They extend normal Petri nets with data types, allowing tokens to carry data. CPNs have developed a rich body of theoretical results that permit analysing dynamic properties of the systems, like boundedness (number of tokens a net may have), invariants (properties that hold true in all execution paths), transition persistence (conflicts) or reachability of certain states [18]. Many of these properties rely on the *occurrence graph*, a representation of the state space that can be model-checked [7]. The CP-nets community has developed a number of tools – CPNTools [19] being the best known one – with a high level of maturity that makes them usable for industrial projects

If we represent a M2M transformation as a CPN, we can use these analysis properties to determine many properties of interest about a QVT-R transformation, beyond the capabilities of current tools. For example, we can check whether the transformation terminates given a starting model by checking if there are terminal states in the net's occurrence graph. It is also interesting to know if the transformation definition may yield different results. This is called confluence, and can be detected if, in addition, the terminal state is unique. A QVT-R transformation may not be confluent if it contains relations dealing with overlapping concerns in different ways (e.g., they produce the same kind of object, but with some difference), and having a conflict: applying one relation disables the other, or makes its result to be different, so that the final result depends on the execution order. This situation can be detected by analysing transition persistence. Finally, we may wish to investigate correctness properties of the transformation, by checking if certain structures are produced or not in the target models. This can be done by model checking the net's occurrence graph.

In the present work, we profit from the theory and tools developed for CPNs by providing a semantics for QVT-R in terms of CPNs. Our semantics covers the transformation and check-only scenarios, and extends the semantics given in the standard to cover model matching [24,27] as well. The use of CPNs opens the door to interesting analysis possibilities, and builds a bridge between the MDE and the Petri nets communities. On the practical side, we leverage CPNTools for the execution, debug and analysis of QVT transformations, contributing to increase existing tool support for QVT-R. The explicit and visual nature of CPN models allows debugging and validating the transformation execution graphically, while their executable semantics contributes to better understand the standard. In addition, the analysis capabilities of CPNTools permit verifying the transformations. We also report on *Colouring* [9], a prototype tool chain we have developed atop the Eclipse Modelling Framework (EMF) [42]. The tool provides automatic translation of QVT-R specifications, meta-models and models into the input format of CPNTools, allowing the execution, debugging, verification and validation of transformations. Moreover, the results of the transformation are translated back and shown to the user as models and trace models. In particular, we support the visualization of the generated models, as well as the trace models and disconformities found by check-only transformations.

This paper extends our previous work [10] by covering the check-only and model matching scenarios, by handling meta-models with inheritance of attributes and associations, by detailing the compilation procedure, by providing comprehensive tool support that includes the generation of EMF models from the final state of the net and a rich visualization of the transformation results, and by a comparison with other QVT-R implementations [2,31,33,45]. Our tool is available at [9], where we have also included a repository of example QVT-R transformations.

The paper is organized as follows. Sections 2 and 3 give an introduction to QVT-R and CPNs. Section 4 shows the compilation of QVT-R into CPNs for the transformation scenario, followed by the compilation for the check-only and model matching scenarios in Section 5. Next, Section 6 presents our supporting architecture. Section 7 illustrates the use of CPNs for verification and validation of transformations. Section 8 discusses related research, providing a feature-based comparison of current QVT-R tools. Finally, Section 9 ends with the conclusions and lines of future work. The paper includes an appendix detailing the compilation procedures.

2 QVT-Relations

QVT-R is the highest-level of abstraction language of the OMG standard for Query/View/Transformation [37]. It has a declarative nature and a dual graphical/textual syntax. A QVT-R transformation is made of relations with two or more domains (usually two). Domains are described by patterns made of a set of variables and constraints – similar to UML object diagrams – that the model elements to which they are bound must satisfy to qualify as a valid *binding* or *occurrence* for the pattern. They also have a flag to indicate whether they are *checkonly* or *enforce*. Models of enforced domains may be modified in order to satisfy the relations, whereas models of checkonly domains are inspected to check for disagreements but cannot be modified.

Transformations have a direction. When they are applied in the direction of an enforced domain, the models of that domain may be modified to obtain a model that, together with a given model from the other domain, satisfies the transformation specification. If a transformation is applied in the direction of a checkonly domain, the execution engine must report the locations where the model does not conform to the transformation, but cannot modify the model. The scheme of these standard scenarios is shown in Fig. 1. In the transformation scenario, the most common situation (and the one most widely supported by tools [21,26,31]) is that the target model is initially empty, and then gets populated by the transformation. We call this scenario *batch*. In the *incremental* scenario, the target model may already exist and gets updated upon changes in the source model. In this paper we do not tackle the incremental scenario, which we leave for future work.

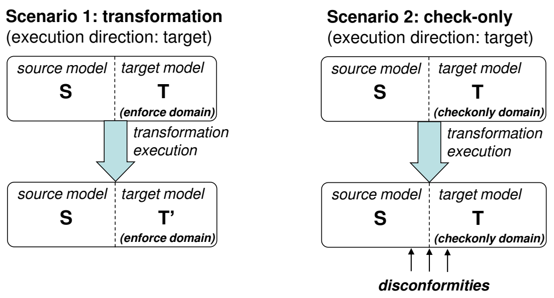


Fig. 1 Usage scenarios of QVT-R specifications: model-to-model transformation and check-only.

A relation is satisfied or *holds* in the source-to-target direction if, for each valid binding of the source domain variables to elements in the source model, there is a valid binding of the target domain variables to elements in the target model. Besides, relations can include *when* and *where* clauses. The former express conditions under which a relation needs to hold. They usually refer to other relations to which they pass the value of bound variables (i.e. model objects bound by the current relation). Thus, a relation only needs to hold in bindings which make the *when* condition hold. *Where* clauses can also contain invocations to other relations using bound variables. In this case, the satisfaction of the relation requires the satisfaction of the invoked relations in all bindings containing the passed variables. Finally, relations can be *top* or *non-top*. After executing a transformation all top-level relations need to hold, and hence the models of the enforced domains may be changed to satisfy the relations. Non-top level relations only need to hold when invoked from the *where* section of other relations.

QVT-R supports Check-Before-Enforce (CBE) semantics [37], which allows relations to reuse objects (instead of creating them) if they already exist. Transformations may declare *keys*. A key is a statement specifying a number of attributes and references for a given class, acting as unique identifiers for objects of that type. These are used by the CBE semantics to decide whether an object exists and can be reused, or must be created.

Throughout the paper we will use an example transformation from a subset of UML class diagrams into relational database schemas. For the sake of comprehension, the example is a simplification of the one given in the QVT standard [37]. The meta-models for both languages are shown in Fig. 2. The left UML meta-model declares `Packages` made of `Classes` with typed attributes. `Packages` and `Classes` have a `kind` attribute, used to mark whether they are persistent[1]. The RDBMS meta-model to its right defines `Schemas`, which contain `Tables`, and these contain `Columns` of a given type.

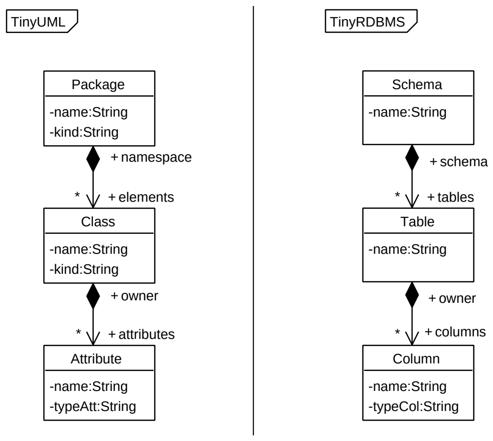


Fig. 2 Source and target meta-models for our example.

The transformation is shown in Listing 1 in textual syntax. It defines three relations, two of them being top-level, and all of them enforcing the RDBMS domain. The first relation demands that for each persistent package

[1] For simplicity, by now we intentionally avoid using class inheritance in the meta-models. We will show how our method handles inheritance in Section 4.6.

in a UML model, there is a schema with the same name (given by the n String variable) but preceded by the prefix 'S'. The second one states that for each persistent class there must be a table with the same name preceded by 'T'. In this case, the *when* section demands this relation to hold only if relation *PackageToSchema* holds for the package and schema containing the class and table. In addition, the *where* clause asks the *AttributeToColumn* relation to hold for the class and the table, which in its turn requires that for each attribute of the class there is a column with the same name and type.

```
transformation umlToRdbms(uml:TinyUML,
                          rdbms:TinyRDBMS) {
  -- maps each package to a schema --------------
  top relation PackageToSchema {
   n: String;
   checkonly domain uml pack:Package {
     name=n, kind='persistent'
   };
   enforce domain rdbms schema:Schema {
     name='S'+n
   };
  }
  -- maps each persistent class to a table ------
  top relation ClassToTable {
   n: String;
   checkonly domain uml class:Class {
     name=n, kind='persistent',
     namespace=pack:Package{}
    };
   enforce domain rdbms table:Table {
     name='T'+n,
     schema=schema:Schema {}
   };
   when { PackageToSchema(pack, schema); }
   where { AttributeToColumn(class, table); }
  }
  -- maps each attribute to a column ------------
  relation AttributeToColumn {
   n, t: String;
   checkonly domain uml class:Class {
     attributes=att:Attribute { name=n, typeAtt=t }
   };
   enforce domain rdbms table:Table {
     columns=column:Column { name=n, typeCol=t }
   };
  }
}
```

Listing 1 Example QVT-R transformation.

2.1 Usage Scenarios and Semantics

As stated before, we can use a QVT transformation in different scenarios. Its execution in the direction of a domain marked as enforce makes it possible to use the transformation for forward (or backward) transformation. We call this scenario *transformation scenario.* For simplicity we assume two domains, and call the domain towards which the transformation is executed the *target domain*, and the other one the *source domain.*

The standard prescribes the execution of the transformation scenario by the compilation of the QVT-R specifications into QVT-C [37]. This latter is a language that relies on the creation of traces to guide the transformation, with a mechanism similar to triple graph grammars [15,39]. Roughly, QVT-C generates one mapping type for each relation, which contains a reference to each object in the relation domains. When executing the transformation, the mechanism creates instances of these traces together with new elements in the enforced domain, for each binding of the variables in the source domain. Hence, *for all* occurrences of the source part of the relation pattern in the source model, an occurrence of the target elements of the relation should *exist* in the target model, together with a trace relating them.

Fig. 3 shows to the left an example TinyUML model. The right of the same figure shows the transformation of this model into a TinyRDBMS model, together with the generated traces. As the initial model contains a persistent package, the relation *PackageToSchema* is enforced, creating a new schema and the corresponding trace. Next, the relation *ClassToTable* is enforced because it is top-level and the *when* condition holds. In this way, the table and its trace are created, and the relation *AttributeToColumn* is explicitly invoked with the class and table as parameters. Enforcing the latter relation creates the column and its trace. At this point, the execution ends as there are no more relations to enforce.

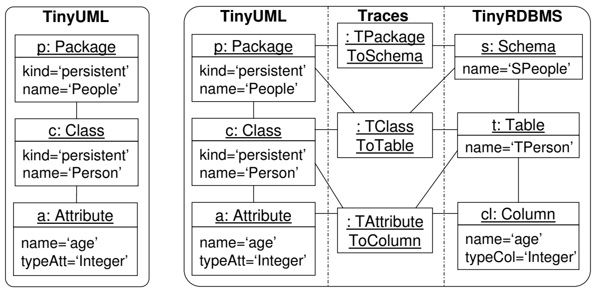


Fig. 3 Initial model (left). Result of applying the transformation in the `TinyUML` $\rightarrow$ `TinyRDBMS` direction (right).

This example illustrates the generation of a target model from scratch. In the *update* or *incremental* scenario, a previously generated target model may need to be updated due to changes in the source model. Finally, in the *synchronization* scenario, any of the models involved in the transformation may need to be updated due to changes in the other. In this paper we cover the transformation scenario, but not the update or the synchronization one, which are left for future work.

The QVT standard also provides another usage scenario called *check-only.* In this scenario, the transformation is executed in the direction of a check-only domain, and it is checked whether the target model is consistent with the source one, but no model is modified. For this purpose it is checked whether all top-level relations are satisfied forwards. As an example, Fig. 4 shows two models where we want to check whether the target is consistent with the source. The check-only procedure returns that this is not the case because relation *AttributeToColumn* is not satisfied in objects c, t and a. This is so because this relation demands a column in the table with the same name as a and type `Integer`, which does not exist. As *AttributeToColumn* is called in the *where* clause from *ClassToTable*, this second relation does not hold either. Finally, as *ClassToTable* is top, we can conclude that the models do not satisfy the specification.

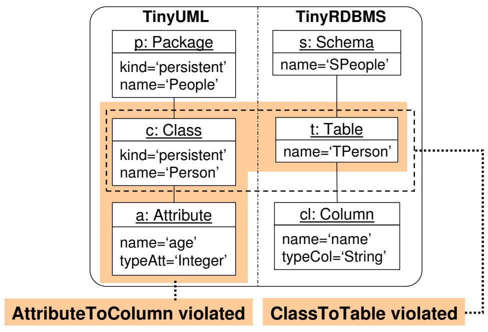


Fig. 4 Disconformity obtained in a check-only scenario.

In addition to the generation of QVT-C, the standard provides an informal procedure in pseudocode that describes how to solve the check-only scenario.

All the aforementioned scenarios assign a direction to the transformation, which we can interpret as the source model being the "main" model for the scenarios. Thus, we first look for all occurrences of the source pattern of the relations in the source model, and for each one of them it is enough to find one occurrence of the target pattern. Hence, conceptually, the source domain of the relations is quantified with a *for all* (so that all occurrences or bindings of the source pattern are sought), while the target domain is quantified with an *exists* (one valid binding of the target pattern should exist). However, the practice of MDE has made evident the need to align models through their comparison in order to check for similarities or with the purpose of merging them [27, 28]. In this direction, dedicated languages such as the Epsilon Comparison Language (ECL) [24] have been developed to populate the traces between models according to a set of comparison criteria. We call this scenario *model matching* or *model comparison*, and in this paper we propose extending the QVT-R semantics to cover it. All models in this scenario are primary and none is modified, as the objective is to trace each combination of occurrences of the source and target patterns according to the relations in a transformation. Fig. 5 shows the schema of this scenario. Conceptually, the source and target models are the "main" models for the scenario, whereas the trace model is the "target" (i.e. for all occurrences of the source and target domain patterns of a relation, a trace should exist).

As an example, Fig. 6 shows the result of a model comparison process which relates each occurrence of the source and target patterns of each relation. In this scenario, the source and target models are given, and the operational mechanism creates the traces. In particular,

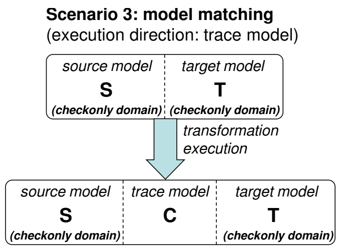


Fig. 5 Schema of the model comparison scenario.

we find one occurrence of each relation. The class is not traced with the table *t1* because relation *ClassToTable* requires relation *PackageToSchema* to hold for the container package and schema, which is not true. If the schema *s1* had the same name as the package with the prefix 'S', then they would have been traced, as well as the class and *t1*.

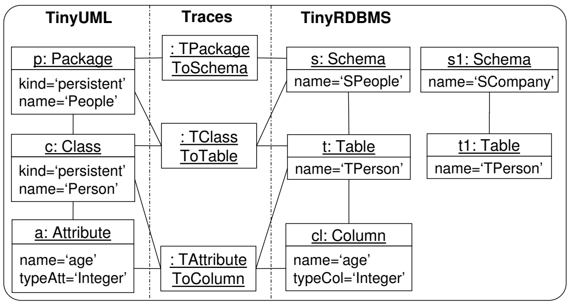


Fig. 6 Result of comparing two models: Unique matching.

Fig. 7 shows another example where the comparison between the source and target models yields several matches of the same relation. In particular, both packages can be matched to each of the two schemas, and hence four traces are created.

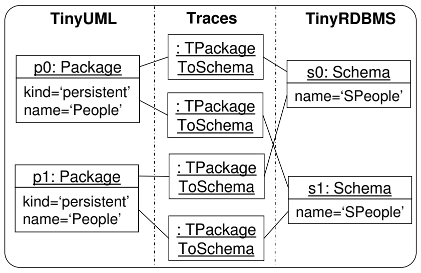


Fig. 7 Result of comparing two models: Multiple matchings.

Once we have revised QVT-R and its usage scenarios, we next give an overview of CPNs as the semantic domain to which we will map QVT-R transformations.

3 Coloured Petri Nets

CPNs is a popular formalism for describing concurrent systems, which is both state and action oriented. Here we give a brief introduction, see [18,19] for more details.

A CPN model can be seen as a bipartite graph with two kinds of nodes: *places* and *transitions*. The former contain tokens, which collectively represent the state of the net. Places are depicted as ovals with the name inside. Transitions model actions and are depicted as labelled rectangles. Places can be connected to transitions, and vice versa, by means of labelled *arcs*. As an example, Fig. 8 shows to the left a CPN with two places (`Package` and `Schema`) and one transition (`PackageToSchemaSimplified`). The net models a simplified version of the relation *PackageToSchema*.

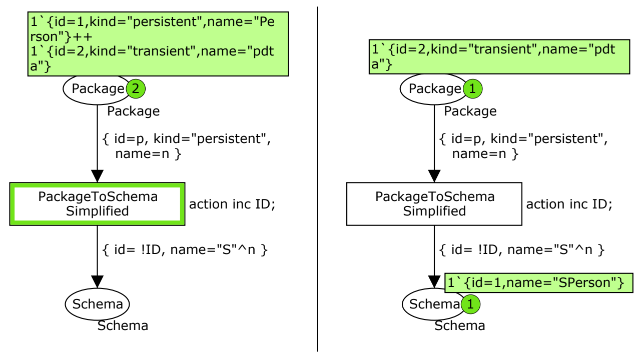

1`{id=1,kind="persistent",name="Person"}++
1`{id=2,kind="transient",name="pdta"}
Package 2
Package
{ id=p, kind="persistent", name=n }
PackageToSchema Simplified
action inc ID;
{ id= !ID, name="S"^n }
Schema
Schema
1`{id=2,kind="transient",name="pdta"}
Package 1
Package
{ id=p, kind="persistent", name=n }
PackageToSchema Simplified
action inc ID;
{ id= !ID, name="S"^n }
1`{id=1,name="SPerson"}
Schema 1
Schema

Fig. 8 Example CPN model (left). Net after firing the transition `PackageToSchemaSimplified` (right).

Each place has a data type defining the kind of data it can contain. The name of the data type is usually shown next to the place (`Package` and `Schema` in Fig. 8). Data types, called *colour sets*, are declared in a language based on Standard ML, called CPN-ML [19]. The language allows declaring simple colour sets – like unit, boolean, integer, string and enumerated – and compound ones – like product, record, list, union and subset. Listing 2 shows the declarations for the example of Fig. 8, which include two records called `Package` and `Schema`. The former contains three fields (`id`, `kind` and `name`) of type integer and string, while the latter contains two fields (`id` and `name`). The listing also declares one global variable `ID`, as well as two variables `p` and `n` which actually appear in the net arcs and are used in the binding process for firing the transition, as explained next.

```
colset Package= record
   id: INT*
   kind: STRING*
   name: STRING;

colset Schema= record
   id: INT*
   name: STRING;

globref ID=0;
var p: INT;
var n: STRING;
```

Listing 2 Colour set declarations for the net in Fig. 8.

The state of a CPN is called its *marking*, and consists of a number of tokens located in the different places. Each token contains data according to the colour set of the place where it resides. Places contain multi-sets of tokens (i.e. sets where element repetition is allowed). In the example, the `Package` place contains two tokens. The number of tokens in a place is indicated in a circle near the place, whereas the cardinality of each token in a multi-set is shown explicitly before the element value (e.g. `1 ` {...}`).

Transitions are the dynamic elements in the net. A place connected through an arc to a transition indicates that the transition, if fired, will remove tokens from the place. These places are sometimes called “incoming” places for the transition. Similarly, an arc from a transition to a place indicates that firing the transition will put tokens into the place. These places are sometimes referred as “outgoing”. Arcs are labelled with expressions used to select the tokens from the incoming places, or to give values to the produced tokens.

Transitions have a guard, shown between brackets, which is a boolean expression involving variables typed on the colour sets. They may also have an associated action which is executed whenever the transition fires. The transition in the example does not contain guards. If it fires, it will execute its action `inc ID;` which increments the value of the global counter `ID`, and then it will remove one token from `Package` and put one token into `Schema`.

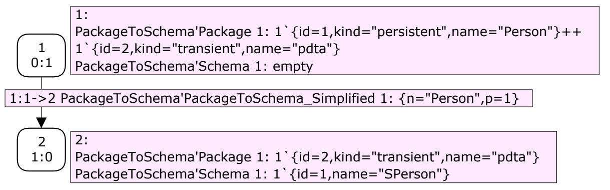
A *binding* of one transition is an assignment of values to the variables in the incoming arcs and the guard. A transition is *enabled* if there is a valid binding for it. This includes checking: (a) if the incoming places have at least one token each able to bind the variables appearing in the incoming arcs, (b) the variables in the guard are bound, and (c) the guard expression evaluates to true. In the example, the transition is enabled[2] because the arc from `Package` demands one token with value “persistent” in its field `kind`, which exists. Thus, the transition is enabled with the binding $b_1 = \langle n = \text{“Person”}, p = 1\rangle$.

An enabled *step* is a finite, non-empty multi-set of bindings enabling transitions. An enabled step can *occur*, whereby some of the enabled transitions fires, changing the marking of the enabled transitions by the multi-set. In particular, the tokens needed to bind the incoming arcs of the transitions are removed, while tokens are created in the output places according to the expressions of the outgoing arcs. In our example, the only enabled step is made of the binding b_1 shown before. Firing the transition removes one token from `Package`, and adds one token to `Schema` having as *id* the value of variable `ID` and as name the name of the removed token preceded

[2] CPNTools depicts enabled transitions by highlighting them in green colour.

by 'S' (string concatenation is written "^" in CPN-ML). The result of the firing is shown to the right of Fig. 8. The resulting net has no available binding as the incoming arc to the transition demands a token with value "persistent" in its field `kind`, which does not exist.

In addition to execution, CPNs have developed a rich body of theoretical results enabling analysis. Some of them are based on the occurrence graph, which is a representation of the set of reachable markings in the form of a graph [18]. As an example, Fig. 9 shows the reachability graph of the net in Fig. 8, automatically computed by CPNTools. The graph shows that from the initial marking (upper node of the graph), the model can only reach one possible state (lower node in the graph). The graph nodes show the marking of the state they represent, and the graph edges stand for fired transitions and their binding. Section 7 will use some CPN analysis techniques to verify properties of transformations.



```
1:
PackageToSchema'Package 1: 1`{id=1,kind="persistent",name="Person"}++
1`{id=2,kind="transient",name="pdta"}
PackageToSchema'Schema 1: empty
```

```
1:1->2 PackageToSchema'PackageToSchema_Simplified 1: {n="Person",p=1}
```

```
2:
PackageToSchema'Package 1: 1`{id=2,kind="transient",name="pdta"}
PackageToSchema'Schema 1: 1`{id=1,name="SPerson"}
```

Fig. 9 Reachability graph for the net to the left of Fig. 8.

CPNTools offers additional hierarchical and modular modelling capabilities to tackle scalability. A large net can be divided into *pages* that can be connected by means of *Fusion Places* and *Substitution Transitions.* Hence, pages serve as a modularization mechanism favouring comprehension, easy visualization and debugging, and avoid having to draw large nets in a single diagram. Substitution transitions are hierarchical transitions that contain inside a whole page of the net structure. Their use allows a hierarchical decomposition and visualization of the net. Fusion places are references for the *same* place appearing in several pages, and simply cross-reference the place in the different pages. Thus, a fusion place is equated with one or more other places, so that the fused places act as a single place with a single marking. We say that all these places belong to the same fusion set. Altogether, these mechanisms allow the modelling in different levels of abstraction (with the substitution transitions) and using multiple views (with the pages and fusion places).

4 Compiling QVT-Relations into CPNs: Transformation Scenario

After reviewing the semantics of QVT-R and CPNs, one realizes that the transformation scenario of a QVT-R specification can be naturally represented as a CPN model by mapping the QVT relations into CPN transitions, and encoding the models as tokens inside places whose associated coloured sets represent the types in the source and target meta-models. Each transition will be enabled for some binding whenever the corresponding QVT relation needs to be enforced at a given occurrence (i.e. at a valid binding of the variables in the source pattern). Firing a transition amounts to enforce a relation for a valid binding. Thus, the concurrent, non-deterministic nature of CPNs permits modelling several relations to be enforced at different bindings.

In this section we describe the compilation of QVT-R specifications into CPNs for the transformation scenario. In order to output a modular, hierarchical CPN promoting understandability, easy visualization and debugging, we split the net in different pages and levels of abstraction. In particular, we create: three pages with places that store the objects of the source, target and trace models; one page for each relation in the transformation; a high-level view of the transformation with one substitution transition for each relation and places depicting the *when* and *where* dependencies between them; and one page with the needed infrastructure to load the initial marking (i.e. the initial source model) from files.

The next subsections explain the compilation procedure using the running example, whereas the appendix provides the pseudocode formalizing the different transformation steps.

4.1 Compiling the meta-models and the initial model

The first step is to compile the source and target meta-models into colour set declarations. We start assuming that the meta-models do not contain inheritance relationships, as we will explain how to handle inheritance later in Section 4.6. Our compilation generates a *record* for each class and association in the meta-models. The record declares one field for each attribute in the class, plus an additional field *id* to store a unique object identifier. In case of an association, the record contains the identifier of the classes in each association end, as well as the attributes if it is an associative class.

As an example, Listing 2 shows the declarations for classes `Package` and `Schema` in lines 1–8. As we will see in the next subsections, further definitions will be added to store the traces and parameters of the relations.

Next, we create one place for each record, which will hold tokens representing the objects in the source and target models, and store the value of their attributes. We split the places of the source and target meta-models in two different pages to enhance readability. Each place is assigned a fusion set so that it can be referenced from other pages. The details of this step of the compilation are shown in Section A.1 of the appendix.

In addition, a further page is generated to read the initial marking (i.e. the initial source model) from files.

This latter is a technical issue related to CPNTools, as we need to generate one auxiliary transition that will read these files when the transition gets fired, by executing an associated action.

As an example, the upper part of Fig. 10 shows a TinyUML model to be transformed, whereas the corresponding generated places and initial marking are shown below. The fusion set names are depicted inside small rectangles to the lower left of each place. The model contains two classes with equal names (since the metamodel allows this[3]) having one attribute each with equal name but different type.

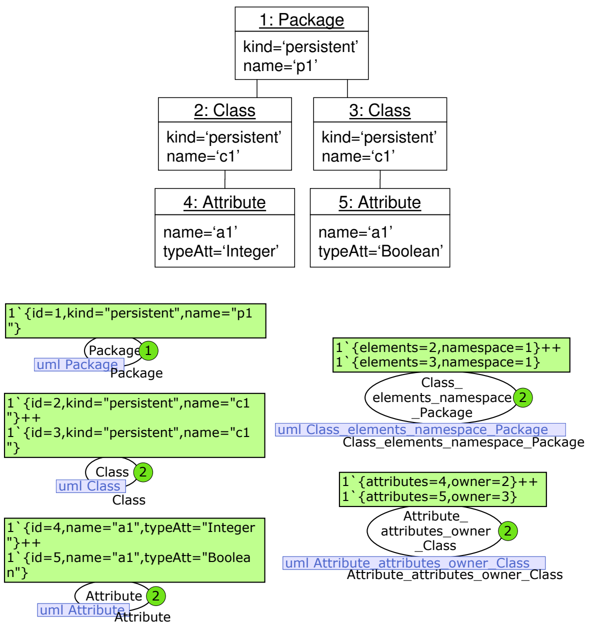

1: Package
kind='persistent'
name='p1'
2: Class
kind='persistent'
name='c1'
3: Class
kind='persistent'
name='c1'
4: Attribute
name='a1'
typeAtt='Integer'
5: Attribute
name='a1'
typeAtt='Boolean'
1`{id=1,kind="persistent",name="p1"}
Package
1
uml Package
Package
1`{id=2,kind="persistent",name="c1"}++
1`{id=3,kind="persistent",name="c1"}
Class
2
uml Class
Class
1`{id=4,name="a1",typeAtt="Integer"}++
1`{id=5,name="a1",typeAtt="Boolean"}
Attribute
2
uml Attribute
Attribute
1`{elements=2,namespace=1}++
1`{elements=3,namespace=1}
Class_elements_namespace_Package
2
uml Class_elements_namespace_Package
Class_elements_namespace_Package
1`{attributes=4,owner=2}++
1`{attributes=5,owner=3}
Attribute_attributes_owner_Class
2
uml Attribute_attributes_owner_Class
Attribute_attributes_owner_Class

Fig. 10 Initial source model (above). Places and initial marking generated from the model (below).

4.2 Compiling the relations

Next, we compile the relations. In this section we restrict to relations with the source domain checkonly and the target domain enforced, and neglect CBE semantics and keys for the moment. We also assume that relations do not have *when* and *where* clauses, which we tackle in Section 4.3.

The scheme of the compilation is shown in Fig. 11. In particular, for each relation we create a transition that will inspect the information in the source domain (hence the arc self-loops) and will produce elements in the target domain. Moreover, we use a tracing mechanism similar to the one in QVT-C. Such a mechanism is implemented by a place that stores the identifier of the transformed elements and prevents executing the transition at the same binding twice. This is actually ensured by invoking a generated ML function in the transition guard.

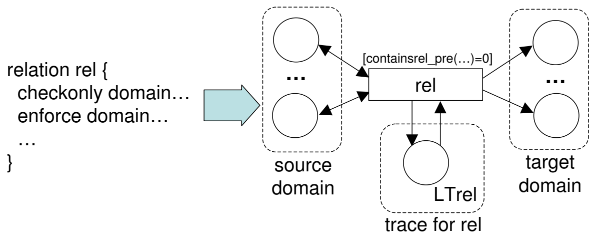


Fig. 11 Schema for the compilation of relations.

More in detail, for each relation, we create a page that includes a transition with the same name as the relation. For each object and link in the relation domains, we add a place from the fusion set corresponding to the type of the element. If the domain is checkonly, the place is connected to the transition with a self-loop, whereas if it is enforced, only the transition is connected to the place. The arc inscriptions contain variables with the same name as in the QVT relation, binding the different fields of the record. Finally, in order to store the traces of the relation, we generate a colour set with the identifiers of all objects appearing in the relation. This conforms to how trace classes are generated from relations in the QVT-R to QVT-C mapping, as it is defined in the standard [37]. We also add to the page one place with a type equal to a list of the aforementioned colour set, storing the traces of the relation (place `LTrel` in Fig. 11). The transition inspects this place in order to check that the identifiers of the objects in the checkonly domain are not in the list, which is checked by the guard function `containsrel_pre(...)`. This avoids enforcing a relation more than once for the same binding. When the transition fires, it creates tokens for the objects in the target domain, adding a new element to the list of traces with the processed objects. The pseudocode of this compilation is given in the Appendix A.2.

We also make the following optimization in the checkonly domain: if the attributes of an object are not accessed, and the object is connected to another through a link `l`, then we do not test if the object is present, but just that there is a link `l`. This is possible as we only need the identifier of the object, and the link `l` already contains it.

Fig. 12 shows the transition generated for relation *PackageToSchema*. The trace place is initialized with one token with the empty list. The read arc takes such list, the guard checks that a record with the identifiers of the involved source objects is not present (function `containsPackageToSchema_pre`), and the write arc adds the record to the list when the transition fires (list concatenation is written "^^" in CPN-ML). The created

[3] The purpose of this somewhat artificial example is to show the effect of the CBE semantics in Sections 4.5 and 7.

schema is given a unique identifier that is calculated by incrementing the global counter `ID` when the transition gets fired. Then, the new value of this counter is written in the tokens through the arc expressions.

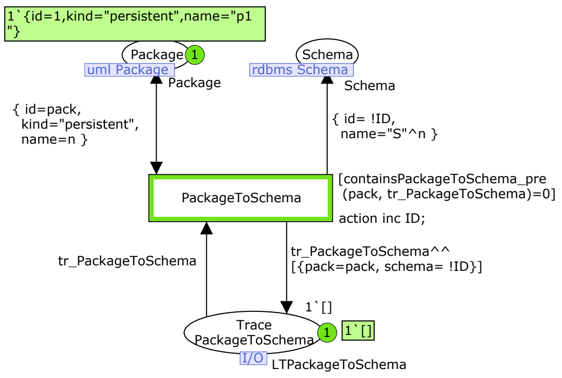

1`{id=1,kind="persistent",name="p1"}
Package
uml Package
Package
Schema
rdbms Schema
Schema
{ id=pack, kind="persistent", name=n }
{ id= !ID, name="S"^n }
PackageToSchema
[containsPackageToSchema_pre (pack, tr_PackageToSchema)=0]
action inc ID;
tr_PackageToSchema
tr_PackageToSchema^^ [{pack=pack, schema= !ID}]
1`[]
Trace PackageToSchema
1`[]
I/O
LTPackageToSchema

Fig. 12 Generated transition from relation *PackageToSchema*.

A QVT transformation can be executed source-to-target or target-to-source. Our method produces two different compilations depending on the chosen direction. This is similar to the approach taken in ModelMorf [33], where Java code is produced instead.

4.3 Compiling the when and where clauses

After generating one transition per relation, we process the *when* and *where* clauses. The latter can include assignment expressions, as well as calls to other relations using as parameters bound objects of the current relation. In order to pass the required objects from the caller to the called relation, we create one intermediate place storing all parameter values, as Fig. 13 shows. Thus, for each relation invoked in the *where* clause (`relA`, ..., `relB` in the figure), we define a colour set with fields corresponding to the passed parameters (`ParamrelA`, ..., `ParamrelB`), and create a place with that type in a new fusion set. We also add an arc from the transition to the place, which writes one token with the given parameter values when the transition fires. Another place in the same fusion set is added to the page of the called relation, and connected to the transition in that page with self-loop arcs. In this way, the passed parameters are not consumed, and the called relation can be enforced as many times as bindings of the source elements exist. Finally, if the called relation receives elements from the target domain as parameters, it does not have to create them, and therefore the arcs reading those objects will be self-loops.

As an example, Fig. 15 shows the transition generated for the relation *ClassToTable*. The upper places correspond to objects and links of the source and target domains. The places for the source domain are read-only

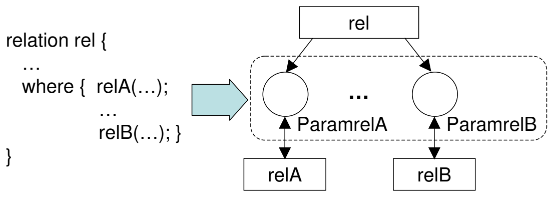


Fig. 13 Scheme for the compilation of *where* clauses.

and hence are connected with self-loops. The places for the target domain are for creation of elements and are connected through outgoing arcs (except the elements received as parameters which are connected through self-loops as well). Please notice that there is no place for *Packages* because we have applied the previously mentioned optimization. The `ParamAttributeToColumn` place is used to pass the two parameters (the class and the created table) to the relation *AttributeToColumn*. The guard in the transition uses the function `containsClassToTable_pre` to avoid transforming the same class twice, as the function checks whether a trace exists for a given class, its package and the corresponding schema. The marking shows the situation after firing the transition once, which creates a table with the name `Tc1`. The transition remains enabled because there is a class that has not been processed, so its firing will create another table also with the name `Tc1`. For the moment we do not take into account the CBE semantics, which would prevent the creation of the second table as there is already one with the same name. We will describe how to modify the net to obtain this behaviour in Section 4.5.

The transition for the relation *AttributeToColumn* is shown in Fig. 16, where the passed parameters are read from the place `ParamAttributeToColumn`. The parameters are not deleted from the place as, in general, a relation may need to be enforced more than once. In addition, we have made the following simplification: if an object comes as a parameter from a *where* clause, its attributes are not accessed, and is connected to some other object, then we do not use the place for the object but the one for the link, since the link stores the object identifier. This is why the places for `Table` and `Class` are omitted in the figure.

The *when* clause is handled by querying the trace places, as shown in Fig. 14. In particular, for each relation *relX* in the *when* clause of a relation *rel*, a self-loop arc reading the trace place `LTrelX` of *relX* is attached to the transition of *rel*. Then, the transition of *rel* is added a guard demanding the existence of a trace in that place for the objects given by the parameters of *relX*. For this purpose, we use the ML function `mem`, which allows checking the membership of a given element in a list. Finally, all write arcs adding tokens to a place of the enforced domain, which correspond to elements passed as parameters in some *relX*, are replaced by self-loops. This is so as these objects already exist and do not have to be created. The details of the compilation of *when* and *where* are presented in Section A.3 of the appendix.

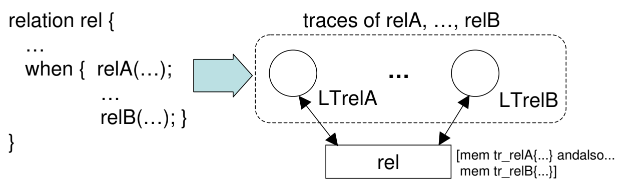


Fig. 14 Scheme for the compilation of *when* clauses.

As an example, the transition generated from *ClassToTable* in Fig. 15 reads the list of traces from place `TracePackageToSchema`. The guard checks that this list contains a record indicating that the package and the schema have already been processed (checked by the function `mem`). Moreover, the place corresponding to the schema is not added tokens (i.e. it has a self-loop from the transition) as the schema is a parameter in the *when* clause, and therefore the transition does not need to create it.

4.4 Adding a high-level view of the transformation

In order to hide the implementation details of each relation, we provide a high-level view of the transformation. This view contains a substitution transition for each QVT relation, referring to the page with the relation details. It also shows the places for the *when* and *where* clauses, so as to depict the execution flow and parameter passing between relations, allowing the identification of dependencies. We can use this high-level view for debugging purposes, following the flow of created objects and traces, as well as the parameters passed between relations. In order to obtain detailed information of how each individual relation works, the user can open the page with the associated transition.

Fig. 17 shows the high-level view of the example, where the top-level relations are depicted with thicker border. Although *ClassToTable* is top-level, it depends on *PackageToSchema* as the latter appears in the *when* clause of the former. Relation *AttributeToColumn* is not top-level and can only be executed when it receives a token with the parameters produced by relation *ClassToTable*. Note how the comments in the QVT transformation are visualized in the net in order to improve understandability.

The target model that results from executing the transformation can be inspected in the page corresponding to the target meta-model. From our example initial model we obtain one schema, two tables with same name, and two attributes. However, this result is not yet consistent with the CBE semantics. The next section explains how to take the CBE semantics into account in the compilation procedure.

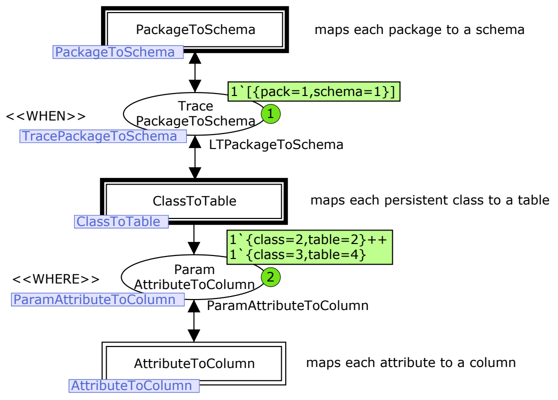


Fig. 17 High-level view of the transformation.

4.5 Check-before-enforce semantics and keys

The CBE semantics ensures that, if an object matching the constraints in a relation already exists in an enforced model, such object will not be newly created. QVT allows defining when two objects are considered equal by means of the *Key* statement. Up to now, the presented compilation has not taken this semantics into account. Even though traces avoided enforcing a relation more than once for the same objects (i.e. for the same binding), we always created objects in the enforced domain instead of reusing them whenever possible. Next we consider such semantics.

The idea is to generate several transitions for each relation. These transitions are mutually exclusive – at most one can fire at any given step – and each one tries to reuse increasingly bigger parts of the enforced domain. Hence, we build a partial order of graphs, the bottom element being the relation parameters (i.e. no reuse), and the top one the graph equal to the enforced domain (i.e. maximal reuse). The keys specify which attributes of an object need to be compared in order to decide whether an object already exists. The handling of traces remains the same, creating a trace each time a transition fires. Recall that the parameters received from a *where* invocation and those passed to another relation in the *when* clause are never created, and hence are reused.

The generated transitions should check if some objects are not present. Negative tests are problematic in CPNs, as the normal arcs test the existence of tokens, not their absence. As inhibitor arcs are not supported by CPNs, we use tokens containing lists of records instead of records. Hence, each place in the enforced domain contains exactly one token with the list of existing objects of a certain type. In this way, testing if an object is not present amounts to ensuring that the corresponding record is not in the list.

Fig. 18 shows the two transitions generated from *PackageToSchema*. The upper one creates a new schema if it is not found on the list `schemas` taken from place `Schema`, actually checked by the function `containsSchema` (which should return 0) in the tran-

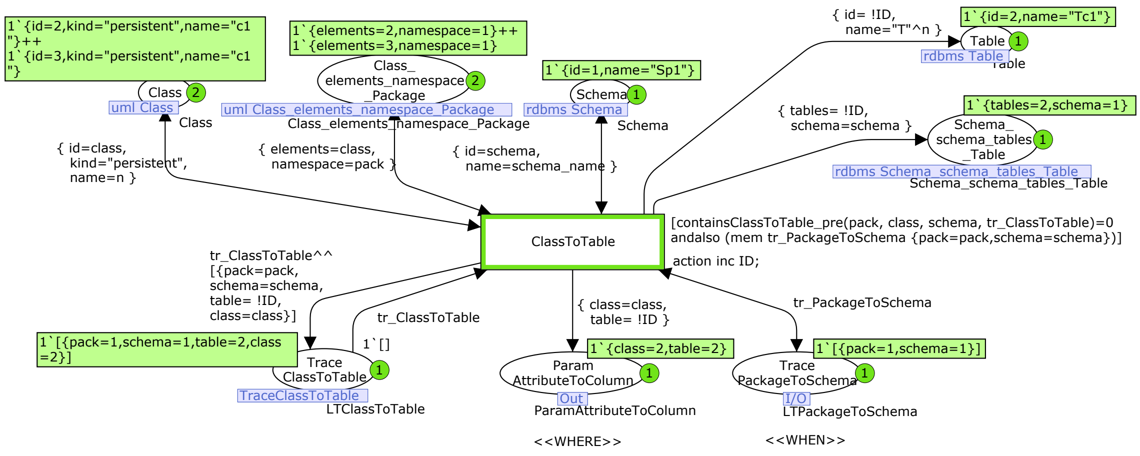


Fig. 15 Generated transition from relation *ClassToTable*.

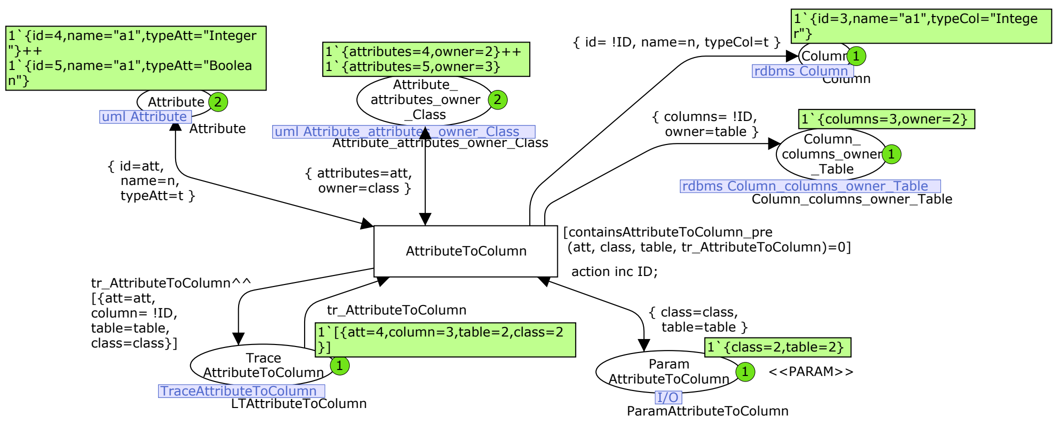


Fig. 16 Generated transition from relation *AttributeToColumn*.

sition guard. The lower one is executed if the schema exists (`containsSchema` returns an index different from 0) and reuses the schema instead of creating it. In this case, the function `getExistingSchemaId` obtains from the list `schemas` the identifier of the schema to be reused, in order to generate a trace for the package and schema.

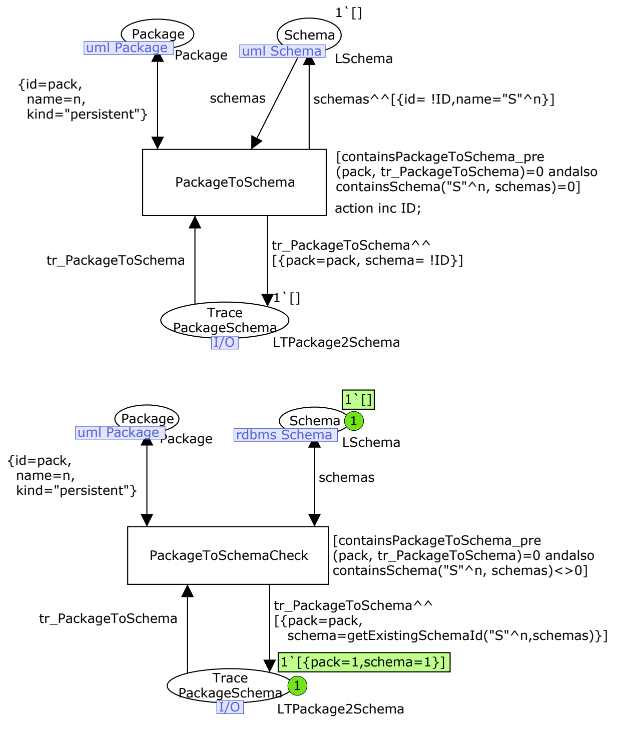
Fig. 19 shows the high-level view of the transformation with all transitions generated by the CBE semantics. The key for the table is its name and schema, and for the column its name and table. The marking shows some traces after executing the net, where only one table and one column are created, in conformance with the CBE semantics. Please note that we currently do not consider possible value clashes in attribute assignments caused by the reuse behaviour of the CBE semantics, that is, we do not detect when the same attribute may be assigned two different values by two different rules, which should be reported as an error according to the specification [37].

4.6 Handling inheritance in the meta-models

The inclusion of inheritance relationships in the meta-models has an impact both on the initial marking generated from models and on the CPN transitions generated from the QVT relations. We will illustrate the way of handling inheritance through a simple transformation example that involves the meta-model shown to the left of Fig. 20. This meta-model defines one class `A` with a child class `B` that inherits all attributes and associations from its parent. The center of the figure shows an instance of this meta-model.

Regarding the compilation of meta-models with inheritance, we create colour sets and places for the meta-model types as explained in Section 4.1, including records for the inherited attributes as well. The generation of the initial marking from a model is slightly different, as we consider that an object with a given type can be seen as an object of any of its supertypes. Therefore we generate additional tokens with the same identifier in the places of the supertypes. In our example, from any object with type `B`, we generate one token in place `B` and another in place `A`, both sharing the same identifier as they refer to the same object. Fig. 20 shows to the right the initial marking generated for the model to its left.

A QVT relation may contain variables of non-leaf classes, which should be bound to objects of the class or of any of its subtypes. The way in which we generate



Package
uml Package
Package
Schema
uml Schema
LSchema
1`[]
{id=pack, name=n, kind="persistent"}
schemas
schemas^^[{id= !ID,name="S"^n}]
PackageToSchema
[containsPackageToSchema_pre (pack, tr_PackageToSchema)=0 andalso containsSchema("S"^n, schemas)=0]
action inc ID;
tr_PackageToSchema
tr_PackageToSchema^^ [{pack=pack, schema= !ID}]
1`[]
Trace PackageSchema
I/O
LTPackage2Schema
Package
uml Package
Package
Schema
rdbms Schema
LSchema
1`[]
1
{id=pack, name=n, kind="persistent"}
schemas
PackageToSchemaCheck
[containsPackageToSchema_pre (pack, tr_PackageToSchema)=0 andalso containsSchema("S"^n, schemas)<>0]
tr_PackageToSchema
tr_PackageToSchema^^ [{pack=pack, schema=getExistingSchemaId("S"^n,schemas)}]
1`[{pack=1,schema=1}]
Trace PackageSchema
1
I/O
LTPackage2Schema

Fig. 18 Two transitions generated from relation *Package-ToSchema* due to CBE semantics.

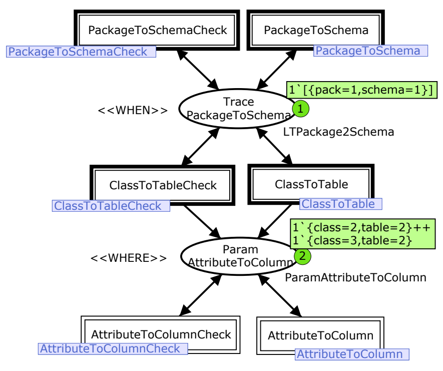


Fig. 19 High-level view considering CBE semantics.

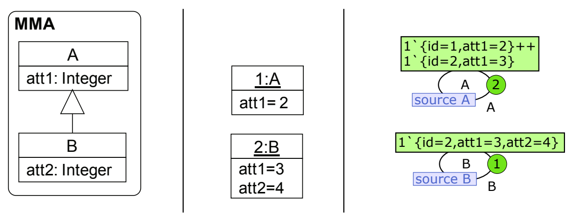


Fig. 20 Meta-model with inheritance (left). Instance of meta-model (center). Generated marking (right).

the marking, adding tokens to the places of every super-type, ensures this behaviour in the checkonly domain. However, in the enforced domain, we have to modify our compilation procedure as follows: whenever we create one object, apart from adding a token to the place of the object's type, we also add one token to the places of the supertypes of the created object.

For instance, Listing 3 shows a transformation between two models conformant to the MMA meta-model shown in Fig. 20. Its unique relation maps each A object to a B object, and thus applies to any object of type A or B in the source model.

```
transformation MMAtoMMA (source:MMA, target:MMA) {
  -- maps each A to a B -------------------------
  top relation AtoB {
    value: Integer;
    checkonly domain source a:A {
      att1=value
    };
    enforce domain target b:B {
      att1=value, att2=value*2
    };
  }
}
```

Listing 3 Example QVT-R transformation.

The generated transition, shown in Fig. 21, reads and transforms objects of types A and B because any object of type B is also stored in place A. Moreover, the created B object is stored in both places B and A2 [4], although in the latter case only with the attributes defined in class A. Please note that since the source and target meta-models are the same, we differentiate the places of the source and target models through the name of the fusion set (source A, target A and so on). Thus, two collections of fusion sets are generated in the example to store the source and target objects separately.

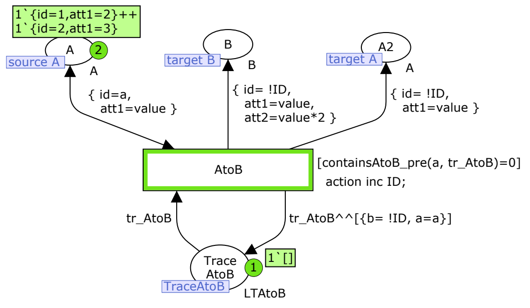
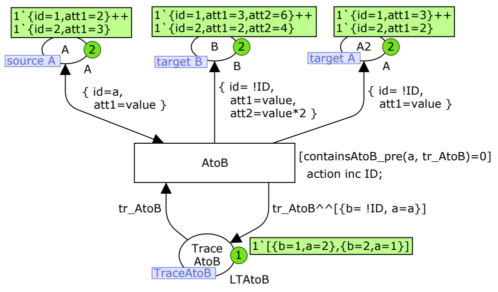


Fig. 21 The resulting CPN net.

Fig. 22 shows the result of executing the net, which creates a target model with two B objects, one for each object in the initial model. Fig. 23 shows the final marking in the form of a model.

Our way of handling inheritance purposely replicates objects and attributes in order to simplify the structure of the generated net. We could avoid data replication, i.e. not storing the attributes of A objects in B objects. However, in such a case, the transitions should inspect the value of inherited attributes in the places of the supertypes, thus increasing the number of read arcs and making the final net less understandable and less efficient.

[4] A2 stores objects of type A, but we use a different name because CPNTools does not allow duplicated place names in the same page.



1`{id=1,att1=2}++ 1`{id=2,att1=3}
1`{id=1,att1=3,att2=6}++ 1`{id=2,att1=2,att2=4}
1`{id=1,att1=3}++ 1`{id=2,att1=2}
source A
target B
target A
{ id=a, att1=value }
{ id= !ID, att1=value, att2=value*2 }
{ id= !ID, att1=value }
AtoB
[containsAtoB_pre(a, tr_AtoB)=0] action inc ID;
tr_AtoB
tr_AtoB^^[{b= !ID, a=a}]
Trace AtoB
1`[{b=1,a=2},{b=2,a=1}]
TraceAtoB
LTAtoB

Fig. 22 Execution result.

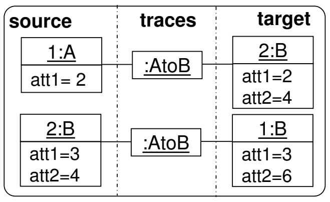


Fig. 23 Resulting models.

5 Compiling QVT-Relations into CPNs: Model Matching and Check-Only Scenarios

In this section we explain our approach to solve the model matching and check-only scenarios. In the former, the aim is generating a trace for each valid binding of the source and target variables of QVT relations. Intuitively, these are the traces that all possible executions of a forward and a backward transformation would generate. In the check-only scenario, we use these traces to check if a relation is satisfied in the forward or backward direction. For this purpose we generate a number of transitions that seek all bindings of the source resp. target domain where a relation should be satisfied (i.e. the relation is "enabled"), and then look if there is a trace for the relation at such bindings (i.e. the relation is "satisfied"). If there is no trace, then the relation is not satisfied, and we report the location where the relation does not hold.

5.1 Model matching scenario

The aim in model matching is creating a trace for each valid binding of the source and target variables of the relations in a QVT specification. The net for model matching is similar to the one for transformation, but the arcs from places both in the source and target domain are self-loops, as in this case both domains are check-only. Hence, transitions only take care of creating traces, but do not create source or target objects. Fig. 24 illustrates the compilation scheme of this scenario.

As an example, Fig. 26 shows the page generated from relation *ClassToTable* in model matching. Firing the transition just adds a new trace to the place `TraceClassToTable`, and passes parameters to relation *AttributeToColumn* through the place

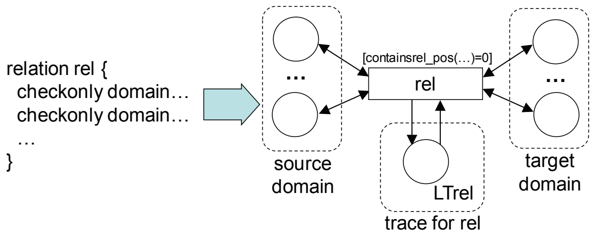


Fig. 24 Compilation scheme for the matching scenario.

`ParamAttributeToColumn`. Please note that the condition on the names of the class and table is checked in the first line of the transition guard. In addition, as before, we generate a high-level view of the net.

As discussed in Section 2, the standard semantics of QVT-R does not consider this scenario. The reason is that model matching demands a *universal occurrence* of both the source and target patterns of relations at the same time, and for each combination, the corresponding trace. On the contrary, the QVT-R semantics demands an *existential* occurrence in a given direction. In this way, in forward transformation, each occurrence of the source pattern demands the existence of one occurrence of the target. Nonetheless, we believe that the model matching scenario could be easily incorporated into the standard and in supporting tools like MediniQVT and ModelMorf, hence enabling the use of QVT-R as a model comparison language [16,24,27].

5.2 Check-only scenario

The check-only scenario does not create source or target elements, but just checks whether two models satisfy a transformation and, if this is not the case, reports which relations are not satisfied and where. Like the transformation scenario, this one also has a direction, and thus it is possible to check whether a target model is consistent with a source model (forward checking) and vice versa (backward checking).

The standard provides a procedure for this scenario which is independent of the existence of traces between the two models. Besides, the standard also proposes a compilation of QVT-R into QVT-C that relies on the creation of traces. Our way to proceed is based on the observation that model matching generates *all* traces that would be generated either in forward *or* backward transformation. Therefore, in order to check a transformation source-to-target, we first perform model matching of the source and target models, and then check that all traces that should exist if we would have performed a forward transformation actually exist. For the missing traces, we collect the identifiers of the objects involved in each non-satisfied relation, and output these as feedback to the user. For this purpose we make use of additional places that store these identifiers as traces.

Altogether, in this scenario we generate the net for model matching, plus one additional transition for each relation which detects valid bindings of the relation pre-condition that are not satisfied. The scheme of these transitions is depicted in Fig. 25. Each new transition is connected with self-loop arcs to the places of elements in the relation pre-condition (i.e. elements in the source domain of the relation, or elements in the target domain that are received as parameters or used in the *when* clause). The transition is also connected to the trace places of any relation in the *when* clause, although we do not show it for simplicity. Finally, we add an "error" place (`LTrel_err`) where the transition writes traces storing the identifier of the objects that belong to a binding of the pre-condition that is not satisfied. The transition guard must check the following two conditions: (i) there is no matching trace that includes the relation pre-condition (so a missing trace is detected), and (ii) the error place does not include the missing trace yet (so the missing trace is not added twice). Note that since check-only transformations have a direction, the pre-condition for the backward case is different from the one in Fig. 25, taking all elements of the target domain instead of the source.

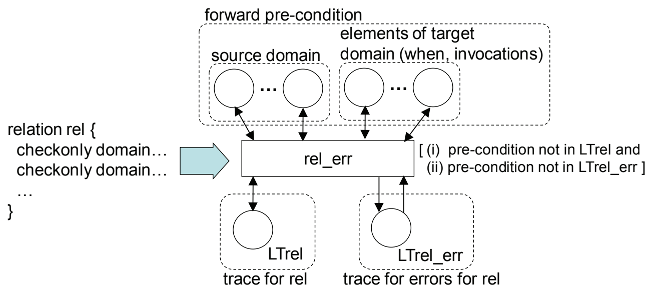


Fig. 25 Transitions detecting missing traces.

As an example, Fig. 27 shows the additional transition generated from relation *ClassToTable* for the check-only scenario, in the `TinyUML` → `TinyRDBMS` direction. All places are read-only except `TraceClassToTable_err`, whose list token gets increased with a new record when the transition fires. Each of these records represents a location where the relation is enabled and should be enforced but it is not. The guard in the transition checks that: (i) there is no trace of relation *ClassToTable* containing the forward pre-condition made of the package, class and schema; (ii) there is not an error trace with these objects yet; and (iii) the package and the schema satisfy the *PackageToSchema* relation, which is checked by looking for a trace of the relation with their identifiers. Note that in contrast to the transitions for the transformation and matching scenarios, this transition does not include a place for the table as this does not belong to the pre-condition of the relation (i.e. the table is the element that the forward transformation should create).

There is a second source of non-satisfaction of relations which comes from the non-satisfaction of their *where* clauses. The previous transitions only detect missing traces, that is, occurrences of the pre-conditions for which no occurrence of the complete graphical pattern is found. However, they do not consider that even if we find the complete pattern, a relation may not hold due to its *where* section.

To solve this problem we generate additional transitions that propagate missing traces up the chain of *where* calls. Thus, if a relation calls in its *where* clause another relation that does not hold, then the caller relation does not hold either. In particular, given a relation, we generate one additional transition for each invocation from its *where* section. These transitions are similar to the previous check-only ones, but they also require that the place for missing traces of the invoked relation contains a trace with the objects passed as parameters in the invocation.

Fig. 28 shows the schema of these transitions, which detect error traces in the invoked relations, and add these traces to their own trace place for errors. Their guard checks that there is a trace which includes all elements in the binding of the source and target of the relation (condition (i)), that there is no error trace with the forward pre-condition objects yet (condition (ii)), and that there is an error trace in some relation called from the *where* clause (`relA` in the figure) using objects in the relation binding as parameters (condition (iii)).

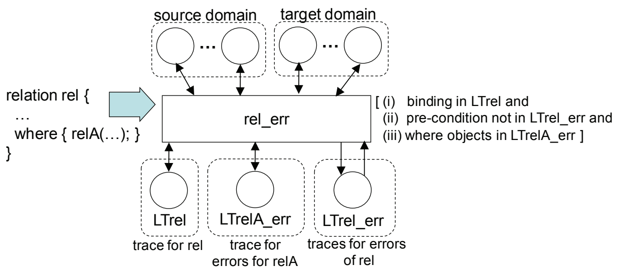
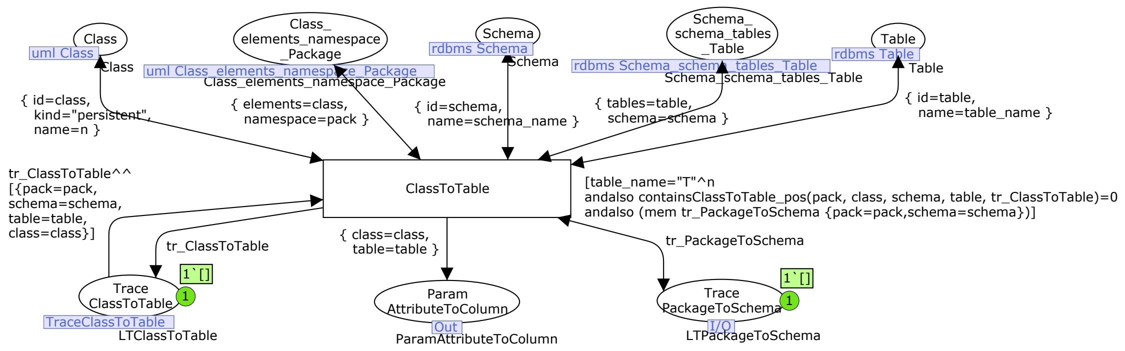


Fig. 28 Transitions propagating missing traces.

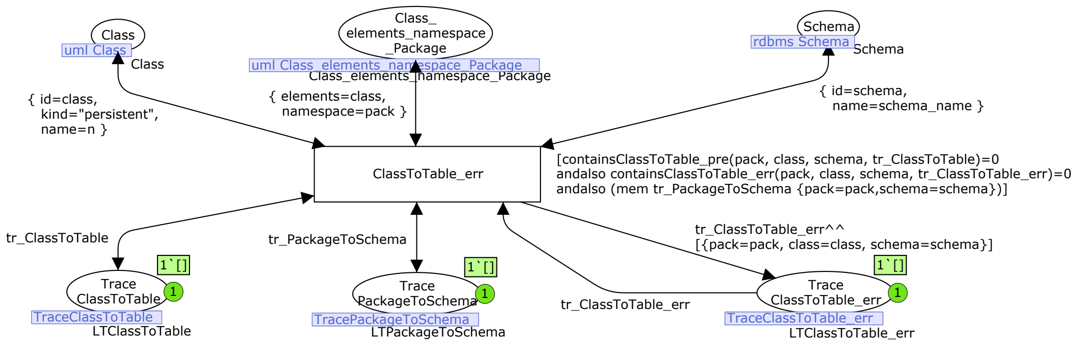
As an example, Fig. 29 shows the transition for relation *ClassToTable* and the invoked *AttributeToColumn*. In contrast to the transition shown in Fig. 27, this also reads the place `TraceAttributeToColumn_err` and checks whether it contains an error trace that involves the class and table (checked by function `contains_where_AttributeToColumn_err`). The transition also has as input the places of the "created" elements, as in this case the transition checks that the relation actually holds and therefore a table exists (checked by function `containsClassToTable_pos`).

Please note that our method reports all bindings where a relation is not satisfied, regardless of whether the relation is top or not. However, for a transformation to be satisfied globally, it is enough that all its top-level relations hold.



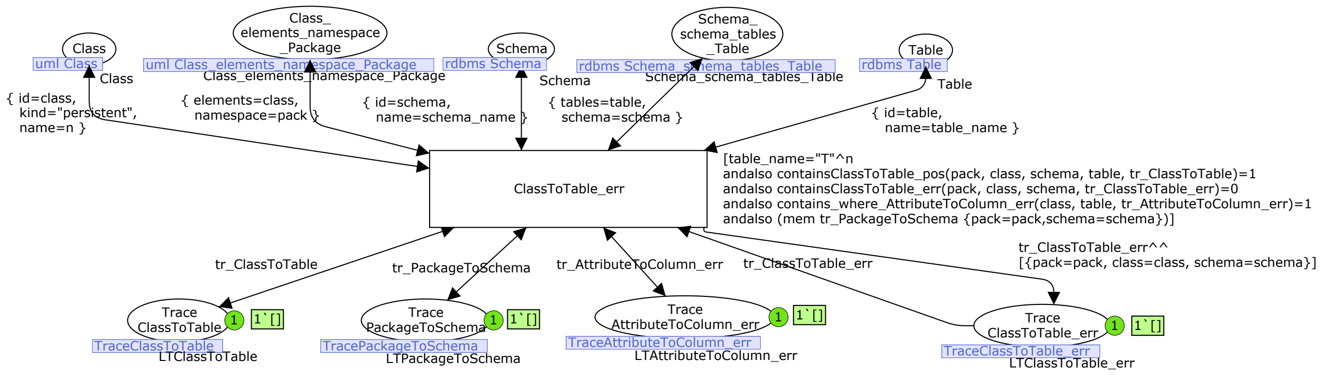
Class
uml Class
Class
Class_ elements_namespace _Package
uml Class_elements_namespace_Package
Class_elements_namespace_Package
Schema
rdbms Schema
Schema
Schema_ schema_tables _Table
rdbms Schema_schema_tables_Table
Schema_schema_tables_Table
Table
rdbms Table
Table
{ id=class, kind="persistent", name=n }
{ elements=class, namespace=pack }
{ id=schema, name=schema_name }
{ tables=table, schema=schema }
{ id=table, name=table_name }
tr_ClassToTable^^ [{pack=pack, schema=schema, table=table, class=class}]
ClassToTable
[table_name="T"^n andalso containsClassToTable_pos(pack, class, schema, table, tr_ClassToTable)=0 andalso (mem tr_PackageToSchema {pack=pack,schema=schema})]
tr_ClassToTable
{ class=class, table=table }
tr_PackageToSchema
1`[]
Trace ClassToTable
1
TraceClassToTable
LTClassToTable
Param AttributeToColumn
Out
ParamAttributeToColumn
1`[]
Trace PackageToSchema
1
I/O
LTPackageToSchema

Fig. 26 Generated matching transition from relation *ClassToTable*.



Class
uml Class
Class
Class_ elements_namespace _Package
uml Class_elements_namespace_Package
Class_elements_namespace_Package
Schema
rdbms Schema
Schema
{ id=class, kind="persistent", name=n }
{ elements=class, namespace=pack }
{ id=schema, name=schema_name }
ClassToTable_err
[containsClassToTable_pre(pack, class, schema, tr_ClassToTable)=0 andalso containsClassToTable_err(pack, class, schema, tr_ClassToTable_err)=0 andalso (mem tr_PackageToSchema {pack=pack,schema=schema})]
tr_ClassToTable
tr_PackageToSchema
tr_ClassToTable_err^^ [{pack=pack, class=class, schema=schema}]
tr_ClassToTable_err
1`[]
Trace ClassToTable
1
TraceClassToTable
LTClassToTable
1`[]
Trace PackageToSchema
1
TracePackageToSchema
LTPackageToSchema
1`[]
Trace ClassToTable_err
1
TraceClassToTable_err
LTClassToTable_err

Fig. 27 Generated check-only transition from relation *ClassToTable*.



Class
uml Class
Class
Class_ elements_namespace _Package
uml Class_elements_namespace_Package
Class_elements_namespace_Package
Schema
rdbms Schema
Schema
Schema_ schema_tables _Table
rdbms Schema_schema_tables_Table
Schema_schema_tables_Table
Table
rdbms Table
Table
{ id=class, kind="persistent", name=n }
{ elements=class, namespace=pack }
{ id=schema, name=schema_name }
{ tables=table, schema=schema }
{ id=table, name=table_name }
ClassToTable_err
[table_name="T"^n andalso containsClassToTable_pos(pack, class, schema, table, tr_ClassToTable)=1 andalso containsClassToTable_err(pack, class, schema, tr_ClassToTable_err)=0 andalso contains_where_AttributeToColumn_err(class, table, tr_AttributeToColumn_err)=1 andalso (mem tr_PackageToSchema {pack=pack,schema=schema})]
tr_ClassToTable
tr_PackageToSchema
tr_AttributeToColumn_err
tr_ClassToTable_err
tr_ClassToTable_err^^ [{pack=pack, class=class, schema=schema}]
Trace ClassToTable
1
1`[]
TraceClassToTable
LTClassToTable
Trace PackageToSchema
1
1`[]
TracePackageToSchema
LTPackageToSchema
Trace AttributeToColumn_err
1
1`[]
TraceAttributeToColumn_err
LTAttributeToColumn_err
Trace ClassToTable_err
1
1`[]
TraceClassToTable_err
LTClassToTable_err

Fig. 29 Generated check-only transition from relation *ClassToTable* and its *where* clause.

6 Tool Support

We have developed a tool chain called *Colouring*, based on Eclipse and EMF [42], which supports the presented approach: it transforms QVT-R specifications and EMF models into the input format of CPNTools, and then transforms back the results of CPNTools into EMF models. If the chosen scenario is model transformation, the user gets back the transformed model in EMF format. If the scenario is model matching, the user is able to inspect the generated trace model, which is visualized as an EMF model that refers to the matched models. In the check-only scenario, the user is informed of the locations where any relation is not satisfied.

Fig. 30 shows a scheme of this architecture. In step 1, the engineer specifies the transformation using the textual format of QVT-R, and the source and target meta-models in *ecore*. In this step he uses common MDE tools and may even use MediniQVT to edit and validate the transformation *syntactically*. In step 2, he chooses the desired scenario (transformation, model matching or check-only), and then our tools generate the necessary input files for CPNTools. In particular, we have built a code generator that parses the QVT-R specification by using the MediniQVT parser [31], and then generates the CPN corresponding to the chosen scenario through JET templates[5] [20]. We have also developed another generator that, given an EMF model, generates a marking in a separate file that is read by the CPN. This has the

[5] Currently, our generator does not consider CBE semantics or optimizations of places. This is left as future work.

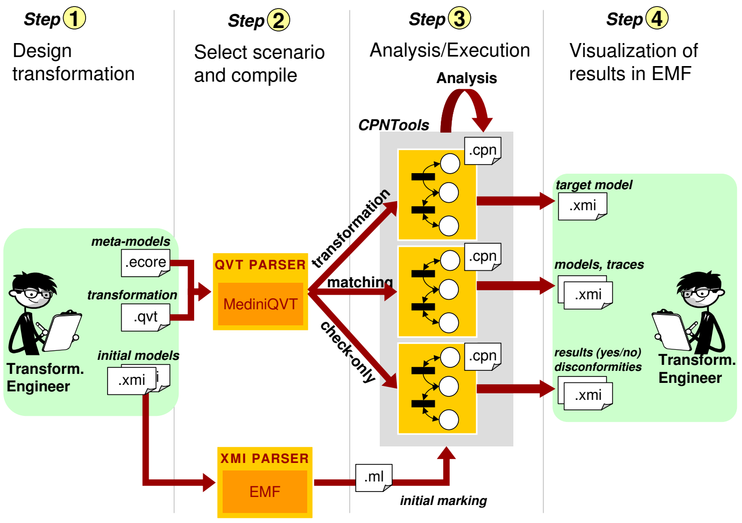


Fig. 30 Architecture of the tool chain.

advantage that there is no need to recompile the whole transformation for different initial models.

Once the CPN is generated, the engineer can analyse it using CPNTools (step 3). The direct use of CPNTools for validation and verification is explained in Section 7. Nonetheless, this step is not necessary if no analysis or debugging is wanted, but just the execution of a transformation for a given scenario.

In step 4, the engineer obtains feedback from the execution in terms of the original EMF models. In case of a transformation, the user is returned the resulting target model in EMF format. This is possible due to a parser we have built that translates the final CPN marking into XMI files. In case of model matching, the user obtains a trace model connecting the related source and target elements. This can be visualized in a unified view as a *Modelink*[6] file, as Fig. 31 shows. In this visualization, the source and target models are shown to the left and right, whereas the trace model is in the middle. By clicking on a trace the engineer can inspect which elements the trace relates, as they are highlighted.

Finally, in check-only scenarios, the engineer is informed of the locations where some relation is not satisfied. Again, the non-satisfied relations are visualized with *Modelink*. Fig. 32 shows the result of a check-only scenario for the two models shown in the left and right panels. In this case, the traces in the middle correspond to occurrences of *relations that are not satisfied*. For instance, the selected (shaded) trace indicates that relation *AttributeToColumn* was not satisfied for the shaded class, attribute (to the left) and table (to the right). Below, the tool summarizes all relations in the transformation and whether they are satisfied or not. A prototype of our tools and some examples can be downloaded from [9].

[6] Modelink is part of the GMT Epsilon project, available at http://www.eclipse.org/epsilon/doc/modelink/

7 Verification and Validation of QVT-Relations

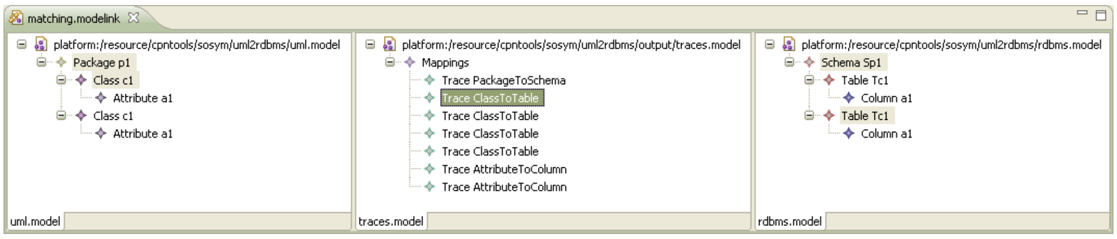
One benefit of compiling QVT-R into CPNs is that we can profit from the large body of analysis methods and tools developed by the Petri nets community. This section presents important verification and validation techniques that are possible once a specification is expressed in CPNs, concentrating in those supported by CPNTools. While verification is concerned with the detection of errors (i.e. is this transformation right?), validation aims at checking whether a transformation behaves as expected (i.e. is this the right transformation?).

7.1 Verification

Many verification techniques for CPNs are based on the computation of the *occurrence graph* [18], a graph-based representation of the space of possible markings. In our case, such graph summarizes all possible execution paths of the transformation given an initial model. Fig. 33 shows the occurrence graph for the running example, considering the forward transformation scenario, CBE semantics, and taking the initial source model shown to the left of Fig. 34. The arrows are labelled with the abbreviated name of the executed relation, and in some cases part of the binding. At a certain stage, several relations can be simultaneously enabled, and the occurrence graph shows all possible orders for their execution.

The nodes without outgoing edges (from 15 to 20) are final states. Each one of them corresponds to a possible transformation result. As an example, Fig. 34 shows in the right two of the possible resulting target models corresponding to the terminal nodes 15 and 16. For clarity, we omit the traces and the source model, but they are also part of the marking.

Next, we describe some interesting transformation properties that can be analysed using CPNTools.

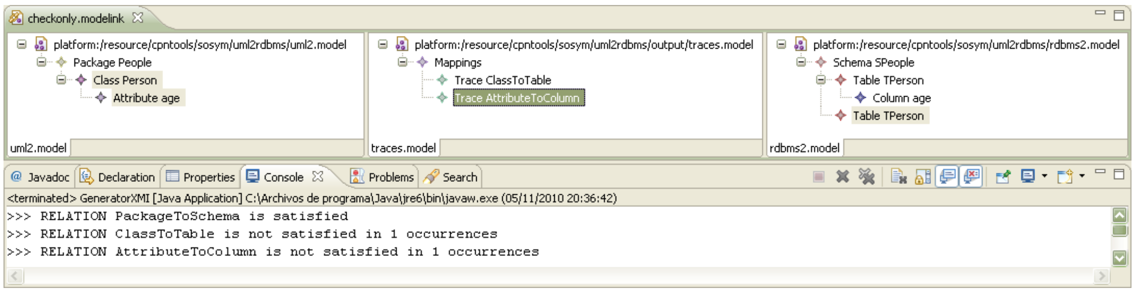


```
platform:/resource/cpntools/sosym/uml2rdbms/uml.model
  Package p1
    Class c1
      Attribute a1
    Class c1
      Attribute a1
```

```
platform:/resource/cpntools/sosym/uml2rdbms/output/traces.model
  Mappings
    Trace PackageToSchema
    Trace ClassToTable
    Trace ClassToTable
    Trace ClassToTable
    Trace ClassToTable
    Trace AttributeToColumn
    Trace AttributeToColumn
```

```
platform:/resource/cpntools/sosym/uml2rdbms/rdbms.model
  Schema Sp1
    Table Tc1
      Column a1
    Table Tc1
      Column a1
```

Fig. 31 Result of model matching. The source and target models are shown to the left and right. The traces are shown in the middle. By clicking on a trace we inspect which elements the trace relates.



```
platform:/resource/cpntools/sosym/uml2rdbms/uml2.model
  Package People
    Class Person
      Attribute age
```

```
platform:/resource/cpntools/sosym/uml2rdbms/output/traces.model
  Mappings
    Trace ClassToTable
    Trace AttributeToColumn
```

```
platform:/resource/cpntools/sosym/uml2rdbms/rdbms2.model
  Schema SPeople
    Table TPerson
      Column age
    Table TPerson
```

```
<terminated> GeneratorXMI [Java Application] C:\Archivos de programa\Java\jre6\bin\javaw.exe (05/11/2010 20:36:42)
>>> RELATION PackageToSchema is satisfied
>>> RELATION ClassToTable is not satisfied in 1 occurrences
>>> RELATION AttributeToColumn is not satisfied in 1 occurrences
```

Fig. 32 Result of check-only scenario: the source and target models are shown to the left and right, and the traces of non-satisfied relations in the middle. By clicking on a trace we inspect the elements belonging to a non-satisfied binding of a relation.

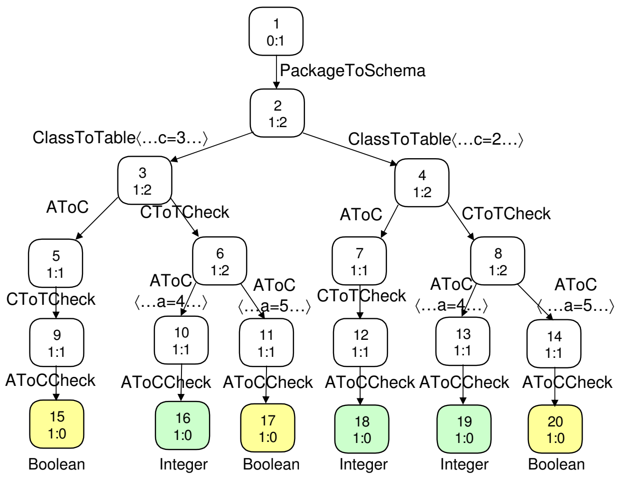


Fig. 33 Occurrence graph for the transformation scenario with CBE semantics.

Confluence. A transformation is confluent if it yields a unique result for each possible initial model. We can investigate confluence for a given initial model by inspecting the terminal nodes of the occurrence graph. In our example we obtain six terminal nodes, which altogether contain two different models (the other four are replicas of these but differ in the object identifiers). Therefore, the transformation is non-confluent. It always creates one table (as both classes have equal name) with one column. However, processing the Boolean attribute first creates a Boolean column (nodes 15, 17 and 20 in the graph), whereas processing the Integer first creates an Integer column (nodes 16, 18 and 19). This is so because the key for attributes only considers their name

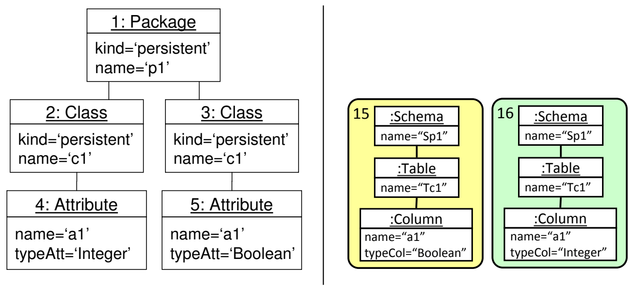


Fig. 34 Initial source model (left). Two possible transformation results for this initial model (right).

and class but not their type, as discussed in Section 4.5. Considering also the column type solves this problem. Note however that CPNs only allows investigating confluence on individual initial models, but not in general.

The occurrence graph shows all theoretically possible results. Nonetheless, a particular tool implementation can still be deterministic by using a particular strategy to select the enabled bindings of relations. Thus, in a non-confluent scenario, two different tools can yield two different results, both of them correct. In our example, the non-confluent behaviour actually causes value clashes in attribute values (i.e. changes in the value of an attribute of a reused object), which should be reported as an error of the QVT specification. We plan to tackle the detection of such errors at run-time (i.e. without performing state space exploration) in future work.

Another source of non-confluence is attribute computation using queries on enforced domains. For instance, if the column name is computed as `name=if (owner.columns->size() = 1) then '_'+n` `else n;` we have non-determinism. This is so because the first column to be processed would be added a prefix `'_'`, and the choice of this column can be non-deterministic. In fact, if the table is added several columns, adding the prefix to the first column is wrong. Since QVT-R is declarative, the expression `name=...` is to be interpreted as an invariant. However, it may yield a different result when evaluated *during* the transformation than *at the end* of the transformation. In this way, if a table is added two columns, then the prefix should not be added to the name of any of them. Thus, "constructive" operational mechanisms would run into troubles and produce a conflict between the transformation scenario and the check-only one. That is, performing a model transformation yields a target model, but testing whether such target model is a correct translation of the source returns failure. In our approach, we forbid attribute computations using queries on enforced domains[7].

Termination. This property is undecidable for graph and term rewriting systems [11]. QVT-R transformations can be non-terminating e.g. due to a recursive relation which creates new elements and passes them to the next step in the recursion in the *where* section. If the occurrence graph is finite and has no cycles, then we can conclude that the transformation always terminates for the given starting model. Thus, our example transformation is terminating for the given initial model.

Relation Conflicts. Two relations are in conflict if they deal with overlapping concerns (i.e., they produce a common set of elements) in different ways, so that executing one may disable the execution of the other or cause its result to be different. It is a source of non-confluence. Transition *persistency* [29] allows discovering conflicts between relations. A transition is persistent if its firing does not disable other enabled transitions, and *weakly* persistent if it may disable itself at a different binding. If a transition is not persistent, this means that it has conflicts with other transitions and may lead to non-confluence. Still, even if there is a conflict, one can have confluence if the paths stemming from the conflict in the occurrence graph lead to a unique final result. Please note that this is similar to the analysis of critical pairs in graph transformation [17].

A conflict in QVT-R may arise for different reasons. First, the execution of some relation may depend on a query on an enforced domain. Second, when the CBE semantics is assumed, two relations may transform different objects into an equivalent one according to the keys (but still with some difference), so that executing the first relation disables the other or makes it produce a different result due to the CBE semantics (hence the final result depends on the chosen execution order). Third, two relations may transform a common set of elements differently (e.g., they produce objects `C` and `D` given an element `A`), and executing one disables the other (e.g., because the produced elements are placed in a collection with maximum cardinality of one). Fourth, when a relation A can be executed only if some other B has not (by placing "*not* $B(\ldots)$;" in A's *when* section). Except in the last case, where explicit conditions are set in the *when* or *where* clauses, relations should be non-conflicting, leading to weakly persistent transitions in the CPN model. Checking for conflicts allows for a fine-grained detection and analysis of the reasons for non-confluence, identifying the conflicting relations causing the problem. All transitions in our example are weakly persistent as none disable others but may disable themselves due to the CBE semantics. Persistency can be efficiently checked using the occurrence graph, and a sufficient condition for persistence exists by statically checking the underlying uncoloured net [35]. This kind of analysis is independent of the initial marking (i.e. of the initial models).

Boundedness and Invariants. We can investigate bounds on the creation of elements, as well as invariants of the transformation. In CPNs, a net is bounded if the number of tokens of all places remains bounded in all possible executions. This analysis is automated by CPNTools and is useful to identify sources of non-termination as well as the maximum number of objects of a certain type that can be created. If the resulting net is unbounded, it means that there is at least one execution path that does not terminate. The converse is not true in general: if the net is bounded, it can still be non-terminating due to cycles, e.g. a recursive call in a *where* clause passing the same parameters it receives.

Invariants are expressions on the marking that remain true in all reachable states. In the context of a transformation, this means expressions that hold true during all steps in the transformation execution. For the analysis of QVT-R transformations, we found useful an invariant consisting in the non-creation of some type of element in enforced domains. Thus, we can check for example whether for a TinyUML model without attributes it is an invariant that no column is generated. These invariants are called synchronization invariants [12] and can be automatically computed.

Model Checking. Sometimes, we are interested in formulating properties about the creation or not of certain patterns in the target model. We can use reachability analysis to investigate whether some structure can be produced in the enforced domain, given an initial model. This procedure can be automated since CPNTools allows expressing properties to be checked on the occurrence graph by means of a CTL-like logic called ASK-CTL [6]. This logic allows formulating queries about states and state changes (e.g. the occurrence of certain transitions). This search is useful to check whether a certain structure is created sometimes or always in each possible result.

[7] MediniQVT does not prevent these queries and may produce target models that together with the source one would not pass the check-only scenario.

For instance, in order to check whether transforming our example initial model always produces a Boolean column, we can use the command `eval_node INV(POS(NF("Has Bool Column", hasColumn))) InitNode`, which returns false as we may obtain an Integer column instead. In the previous command, `InitNode` is the initial marking, `hasColumn` is a user-defined function that checks whether a given marking contains a boolean column, `POS(A)` demands property A to be eventually satisfied, and `INV(A)` demands A to be satisfied in all possible paths. Checking whether sometimes such column is obtained is done through command `eval_node POS(NF("Has Bool Column", hasColumn)) InitNode`, which returns true.

Other interesting properties include whether we always or sometimes obtain the same number of columns as attributes (false in both cases), the same number of tables as classes (false), the same number of schemas as packages (true), or whether a certain relation is always or sometimes executed.

While the previous analysis methods are primarily useful for transformation scenarios, they can be used for model matching and check-only scenarios as well. For example, in model matching we may wish to investigate whether an element is traced more than once by a relation or not traced at all, with the purpose of outlining possible ways of merging two models. This can be checked using boundedness and invariant analysis. In check-only scenarios we may wish to check the causes of failure of a relation, either lack of target objects or failure of a relation called in the *where* section. For this purpose we can visualize the disconformities (see Fig. 32), or perform boundedness and invariant analysis associated to the error places.

Altogether, the compilation of QVT-R into CPNs (instead of into a programming language such as Java, as ModelMorf does) has the advantage that we have the chance to use the analysis methods developed by the Petri nets community, bridging the research done both in MDE and Petri nets.

7.2 Validation of transformations with CPNTools

As we have seen, the theory of CPNs offers techniques to *verify* transformation properties. However, as any other software, transformations should also be *validated* by checking that they behave according to their requirements. In order to validate a transformation we can use CPNTools to perform run-to-completion execution, as well as a step-by-step visual simulation for debugging. Moreover, the multi-view and hierarchical features of this tool permit visualizing the execution flow in the high-level page, and checking the created elements in the pages corresponding to the meta-models and traces.

Fig. 35 shows CPNTools being used to validate the running example transformation. In particular, the user has decided to have three views of the transformation, each one shown in a different panel. The left panel (*Top*) shows the high level view of the transformation and the dependencies between its relations. The panel in the middle (*PackageToSchema*) shows the details of the *PackageToSchema* relation, which has just been fired, resulting in the creation of a schema and its corresponding trace. The right panel (*TinyRDBMS rdbms*) shows the marking for the target model, which contains the created schema. The user may also show or hide the pages containing the source model (*TinyUML uml* in the left column of declarations at the back window), the set of traces (*Traces*), and the other relations in the transformation (second and third tabs in the middle panel). Moreover, as stated in previous sections, we also generate a page to load the initial marking from files (*Load TinyUML uml*).

In addition, similar to breakpoints in programming environments, the user can set *monitors* establishing conditions (e.g. the marking exceeds a certain size, a transition fires a number of times or a place becomes empty) under which some action is performed (e.g. pause the execution or write to a file). They can also be used to encode the OCL constraints of the target language in order to detect their violation. Actually, our generator makes use of monitors to automatically save the final marking into a file, which is subsequently transformed into an EMF representation.

Finally, simulation and verification can be combined using the occurrence graph, as this can be created incrementally and visually inspected. Each node in the graph can show the marking, and it is possible to set the net in the state of a given node.

Altogether, the use of CPNs as target semantic domain has the advantage that we can use existing tools developed for this formalism, such as CPNTools, in order to profit from powerful visual environments for execution, debugging and analysis of transformations.

8 Related Work

Next we review the features of a number of tools for QVT-R, and then discuss other related research.

8.1 Feature-based comparison of QVT-R tools

TROPIC [45] is a tool aimed at debugging QVT-R specifications visually. The TROPIC language is inspired by CPNs although it does not follow their standard semantics, but a tailored semantics more appropriate for transformation where tokens are never consumed by transitions. This fact hinders the use of the standard CPN theory to analyse TROPIC models, but yields simpler nets as e.g. checkonly domains only need read arcs instead of loops. The representation of QVT-R concepts in TROPIC also differs from the one we have presented. For

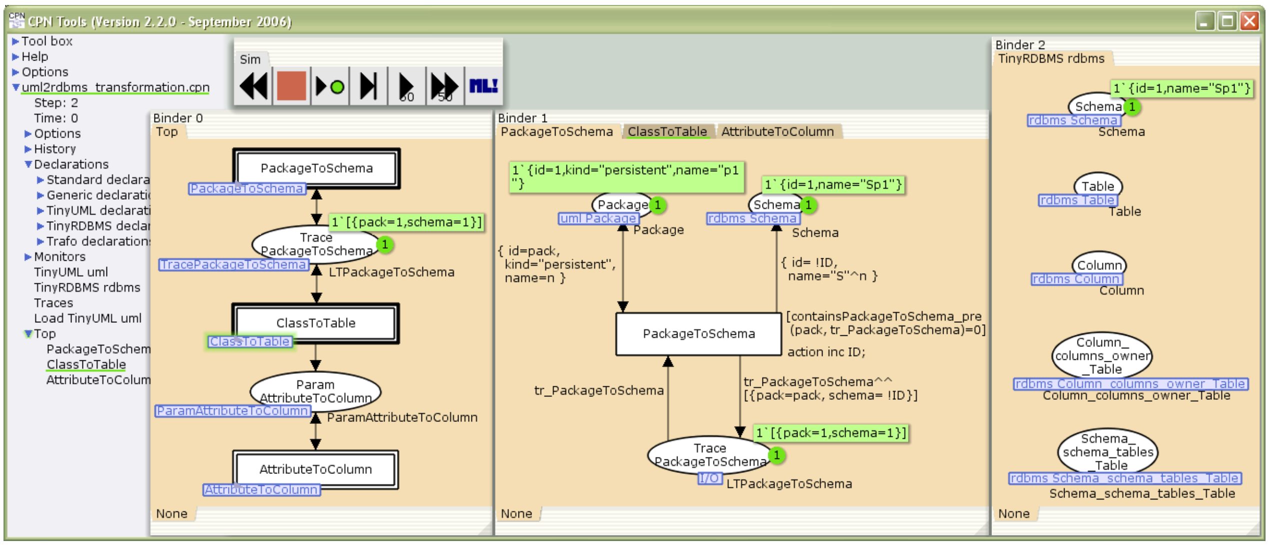


Fig. 35 CPNTools being used to validate the transformation. The left panel contains the high-level view of the transformation, the middle panel contains one QVT relation, and the right panel contains the target model.

instance, they represent classes and attributes in separate places, which produces more complex nets, but at the same time, it avoids data duplication when handling meta-models with inheritance. Finally, to our knowledge, TROPIC does not support CBE semantics, model matching or check-only scenarios.

MediniQVT [31] is a freeware Eclipse plug-in for QVT-R specifications in textual syntax (indeed we have used its parser in our implementation). It includes useful step-by-step debugging capabilities using the standard Eclipse debugger, however it only supports the transformation scenario. ModelMorf [33] is another tool that supports QVT-R, developed by Tata Research Development and Design Centre. The tool has a command line interface and supports both check-only and transformation scenarios. However, lacking user interface, the tool is harder to use than MediniQVT, and it does not provide debugging support. Finally, MOMENT-QVT [2] is an academic prototype built on top of the rewriting logic engine Maude, with very basic functionality only.

Table 1 compares the main features of these QVT-R tools. The first four rows analyse the supported scenarios (transformation, check-only, model matching and incremental update). The next three rows show the support for validation and verification. It is worth mentioning that MediniQVT and TROPIC have dedicated debugging environments, whereas our approach uses CPNTools directly. All approaches generate some artefact for the traces, for example MediniQVT and ModelMorf generate files, while TROPIC puts tokens in certain places. However, ours is the only approach where both traces and relation disconformities can be visualized and explored graphically. The last three columns show advanced concepts like support for OCL, CBE semantics and others.

Table 1 Comparison of different QVT-R tools.

	Colouring	MediniQVT	ModelMorf	TROPIC	MOMENT-QVT
Transformation	√	√	√	√	√
Check-only	√	–	√	–	–
Matching	√	–	–	–	–
Incremental update	–	√	√	–	–
Debugging	√	√	–	√	–
Verification	√	–	–	–	–
Trace display	√	–	–	–	√
OCL support	–	√	√	√	–
CBE semantics	√ [a]	√ [b]	√	–	–
Others	–	–	√ [c]	–	–

[a] Our theory supports CBE semantics, though this feature is under development in our tool
[b] Requires an explicit setting of keys
[c] Sequence and set patterns, transformation inheritance, rule overriding

One can observe that our approach contributes to improve the understanding and support for QVT-R, enables the application of QVT-R to scenarios not considered in the standard like model matching, and allows for the implementation of "low-cost" QVT-R engines allowing execution and analysis (using CPNTools as underlying infrastructure). As drawbacks, our main limitation is the support of complex OCL constructs, which is left as future work. Regarding existing tools, ModelMorf seems the one with more advanced concepts, but also the most difficult to use. TROPIC and MOMENT-QVT have been evaluated by reference to the literature, as they are not available for download. It can be noted that no tool supports nowadays the standard compilation of QVT-R into QVT-C and its subsequent execution.

We have also evaluated the performance of CPNTools used as a transformation tool, by executing the running example transformation with input models of increasing size. Our findings are that CPNTools has good performance for small input models, with an execution time lower than 1 second up to 100 objects. Unfortunately, the performance is exponential on the size of the models, as Fig. 36 shows. Interestingly, the same exponential curve has been reported for MediniQVT in [44] (although with smaller times than ours, which was expected because CPNTools is not a dedicated tool for model transformation). Thus, the development of an efficient tool with a non-exponential performance on the size of the input models remains a challenge for the QVT-R transformation community. Nonetheless, the capabilities of CPNTools to debug, analyse and verify transformations still makes our approach valuable.

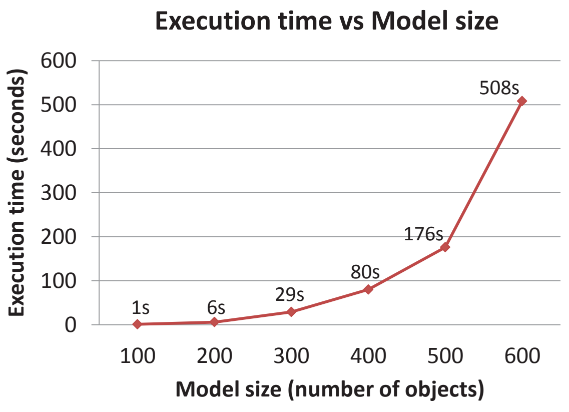


Fig. 36 Execution time for the running example.

Regarding the correctness of our compilation into CPNs, we have run the suite of transformation cases in [9] and [43] with ModelMorf, MediniQVT and Colouring, obtaining the same results in all cases. This does not prove that our semantics is correct, but it provides evidence in favour of such a hypothesis.

8.2 Related research

We now review some existing formalizations of QVT-R, not necessarily supported by tools, with special attention to the features listed in Table 1.

Inspired by the standard compilation of QVT-R into QVT-C, in [15] the authors translate QVT-R into triple graph grammars (TGGs), so that TGGs play the role of QVT-C. Being based on graph transformation [11,39], TGGs have a formal, theoretical basis. They are also supported by well-established tools like MOFLON [34] and Fujaba [13]. Hence, this work serves as a basis for executing QVT-R atop graph grammar-based tools. However, the paper aims at comparing TGGs with QVT-C for the transformation scenario, and does not discuss the CBE semantics or analysis techniques.

Rewriting logic, and Maude [8] in particular, has also been used for expressing the semantics of QVT-R. In [3, 30], the authors formalize QVT-R transformations by using rewriting logic and Maude; however there is no comment about CBE semantics and no discussion on termination or confluence. In [43], the author proposes a formalization for QVT-R check-only transformations based on game theory. This approach is higher-level than the previous ones as it does not give a semantics for pattern matching, but relies on suitable oracles instead. The process of checking the conformance between two models and a transformation specification is described as a game between a verifier and a refuter, which at each state of the game pick bindings in the source and target domains of the relations under evaluation. The approach serves well in clarifying some aspects of the check-only semantics of QVT-R, like the $\forall$-$\exists$ alternation when bindings are sought in the source and target domains, but it does not consider the transformation or model matching scenarios. Similar to our work, we believe that using a formal semantic domain (game theory in [43]) is beneficial as it enables the reuse of theory and analysis methods.

The work in [14] uses OCL for representing the static semantics (well-formedness rules) and provides a translation of QVT transformations into Alloy. Although Alloy permits execution and analysis, no discussion on analysis is given. The approach is similar to our previous work in [5], where we translated QVT-R into OCL and used a constraint solver for execution and analysis. In that case, the kind of possible analyses is different, as they are based on "model finding". For example, we tested whether a transformation is satisfiable, or whether a source model produces a valid target model (i.e. conformant to the target meta-model and its integrity constraints). In our approach with CPNs, the validity of the target model has to be checked by loading and validating the model in the modelling tool, or by setting CPN monitors. However, CPNs allow the visual step-by-step execution and debugging of the transformation, which is not possible with constraint solvers.

In [38], the authors translate QVT-R into QVT-O, so that it is possible to use QVT-O tools like SmartQVT [40] to execute the relations. The approach only supports the transformation scenario, but takes into account the CBE semantics. A similar experiment is proposed in [23], where QVT-R is transformed into the input language of the ATL virtual machine.

Finally, some other works propose extensions to QVT-R. For example, in [36], domain patterns can include numeric inequalities involving attribute values (in addition to attribute assignments) which are solved by a constraint solver. However, the work does not address inter-model constraints relating source and target attributes, which would be very useful for bidirectional transformations. To the best of our knowledge, no previous work has proposed an extension of the QVT-R semantics for its use in model comparison.

9 Conclusions and Future Work

In this paper we have presented an approach for the execution, verification and validation of QVT-R specifications by their compilation into CPNs. The approach supports meta-models with inheritance, *when* and *where* clauses, CBE semantics, as well as transformation, check-only and model matching scenarios. We have shown how to use the occurrence graph to check termination and confluence, how to analyse relation conflicts by transition persistence, and how to test the creation of objects in enforced domains using model checking, invariants and boundedness analysis. We have also demonstrated that CPNTools can be used for the execution, verification and validation of transformations.

In addition, we have developed an EMF-based tool chain, named *Colouring*, which automates the generation of CPNs from QVT-R transformations, and translates the transformation results back as EMF models. We support the visualization and exploration of traces in model matching, and report relation disconformities in check-only scenarios. In general, the results of *Colouring* are in line with those of tools like MediniQVT and ModelMorf.

One limitation of our proposal is the full support for OCL, which would require a complex compilation into ML. Up to now we support arithmetic operations and string concatenation. Complex queries involving negation would require using tokens with lists also in check-only domains. Other features that we currently do not support are: more than two domains in relations, arbitrary expressions of relation invocations in *when* and *where* clauses, set templates, attribute value clashes produced by the CBE semantics, transformation extensions and rule overriding.

With respect to future work, an immediate goal is the technical improvement of *Colouring*, for example to make it an Eclipse plug-in. We are also interested in exploring QVT-R for in-place transformations, as supported by ModelMorf. This would require equating the source and target models by assigning a unique fusion set to the same class in the source and target domains. The extension of our framework to cover update transformation is also under consideration.

Besides, it would be interesting to develop a high-level language to specify the properties to be model-checked. The use of CPNs opens the door to other useful techniques, such optimizing the CPN [12] and translating the optimizations into QVT, or the verification of properties independently of the marking (like termination [1]). We also plan to analyse information preservation, i.e. whether a forward transformation followed by a backward transformation recreates the original model. Complementing our analysis techniques with the synthesis of initial markings for the nets in order to automate transformation testing is also future work.

Finally, it might be interesting to apply our compilation to other languages like ATL or TGGs, so that their semantics is given in terms of CPNs. This would enable the comparison of their semantics on the basis of a common domain, as well as the combination of heterogeneous transformations at the CPN level.

References

1. K. Barkaoui, C. Dutheillet, and S. Haddad. An efficient algorithm for finding structural deadlocks in colored Petri nets. In *APN'93*, pages 69–88, 1993.
2. A. Boronat. *MOMENT: A formal framework for MOdel managemMENT*. PhD thesis, Universitat Politècnica de Valencia, 2007. See also `http://moment.dsic.upv.es/content/view/34/75/`. Last accessed: Nov. 2010.
3. A. Boronat, J. A. Carsí, and I. Ramos. Algebraic specification of a model transformation engine. In *FASE'06*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006.
4. A. Boronat, R. Heckel, and J. Meseguer. Rewriting logic semantics and verification of model transformations. In *FASE'09*, volume 5503 of *LNCS*, pages 18–33. Springer, 2009.
5. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
6. A. Cheng, S. Christensen, and K. H. Mortensen. Model checking coloured Petri nets exploiting strongly connected components. In *WODES'96*, pages 169–177, 1996.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
9. Colouring. `http://astreo.ii.uam.es/~eguerra/tools/colouring/main.htm`. Last accessed: July 2011.
10. J. de Lara and E. Guerra. Formal support for QVT-Relations with coloured Petri nets. In *MoDELS'09*, volume 5795 of *LNCS*, pages 256–270. Springer, 2009.
11. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer, 2006.
12. S. Evangelista, S. Haddad, and J.-F. Pradat. Syntactical colored Petri nets reductions. In *ATVA'05*, volume 3707 of *LNCS*, pages 202–216. Springer, 2005.
13. Fujaba. `http://www.fujaba.de/`. Last accessed: Nov. 2010.
14. M. García. Formalization of QVT-Relations: OCL-based static semantics and Alloy-based validation. In *MDSD today*, pages 21–30. Shaker Verlag, 2008.
15. J. Greenyer and E. Kindler. Comparing relational model transformation technologies: Implementing Query/View/Transformation with triple graph grammars. *Software and System Modeling*, 9(1):21–46, 2010.

16. E. Guerra, J. de Lara, and F. Orejas. Inter-modelling with patterns. *Software and System Modeling*, In press, 2011.
17. R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.
18. K. Jensen. *Coloured Petri nets basic concepts, analysis methods and practical use (Monographs in theoretical computer science)*. Springer, 1997.
19. K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. *STTT*, 9(3-4):213–254, 2007. See also `http://cpntools.org`. Last accessed: July 2011.
20. JET. `http://www.eclipse.org/modeling/m2t/?project=jet`. Last accessed: Nov. 2010.
21. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008. See also `http://www.emn.fr/z-info/atlanmod/index.php/Main_Page`. Last accessed: Nov. 2010.
22. F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *OOPSLA'06*, pages 719–720. ACM, 2006.
23. F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *SAC'06*, pages 1188–1195, 2006. See also `http://www.eclipse.org/m2m/atl/usecases/QVT2ATLVM/`.
24. D. S. Kolovos. Establishing correspondences between models with the Epsilon Comparison Language. In *ECMDA-FA'09*, volume 5562 of *LNCS*, pages 146–157. Springer, 2009.
25. D. S. Kolovos, R. F. Paige, and F. Polack. Merging models with the Epsilon Merging Language (EML). In *MoDELS'06*, volume 4199 of *LNCS*, pages 215–229. Springer, 2006.
26. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Transformation Language. In *ICMT'08*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.
27. D. S. Kolovos, D. D. Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *CVSM'09*, pages 1–6. IEEE CS, 2009.
28. A. Königs and A. Schürr. Tool integration with triple graph grammars - a survey. *ENTCS*, 148(1):113–150, 2006.
29. L. H. Landweber and E. L. Robertson. Properties of conflict-free and persistent Petri nets. *J. ACM*, 25(3):352–364, 1978.
30. F. J. Lucas and J. A. T. Álvarez. Model transformations powered by rewriting logic. In *CAiSE Forum*, volume 344 of *CEUR Proc.*, pages 41–44, 2008.
31. MediniQVT. `http://projects.ikv.de/qvt/`. Last accessed: Nov. 2010.
32. S. J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled*. Addison-Wesley Object Technology Series, 2004.
33. ModelMorf. `http://www.tcs-trddc.com/trddc_website/scripts/project_detail.php?lab=SWRD&project_id=44`. Last accessed: Nov. 2010.
34. MOFLON. `http://www.moflon.org/`. Last accessed: Nov. 2010.
35. A. Ohta and K. Tsuji. On some analysis properties of colored Petri net using underlying net. In *MWSCAS'04*, volume 3, pages 395–398. IEEE CS, 2004.
36. A. Petter, A. Behring, and M. Mühlhäuser. Solving constraints in model transformations. In *ICMT'09*, volume 5563 of *LNCS*, pages 132–147. Springer, 2009.
37. QVT1.1. `http://www.omg.org/spec/QVT/`. Last accessed: May. 2012, 2011.
38. R. Romeikat, S. Roser, P. Müllender, and B. Bauer. Translation of QVT relations into QVT operational mappings. In *ICMT'08*, volume 5063 of *LNCS*, pages 137–151. Springer, 2008.
39. A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
40. SmartQVT. `http://sourceforge.net/projects/smartqvt/`. Last accessed: June 2010.
41. T. Stahl and M. Volter. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.
42. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2^{nd} Edition*. Addison-Wesley Professional, 2008. See also `http://www.eclipse.org/modeling/emf/`. Last accessed: Nov. 2010.
43. P. Stevens. A simple game-theoretic approach to check-only QVT relations. *Software and System Modeling*, In press, 2012.
44. M. van Amstel, S. Bosems, I. Kurtev, and L. F. Pires. Performance in model transformations: Experiments with ATL and QVT. In *ICMT'11*, volume 6707 of *LNCS*, pages 198–212. Springer, 2011.
45. M. Wimmer, G. Kappel, J. Schönböck, A. Kusel, W. Retschitzegger, and W. Schwinger. A Petri net based debugging environment for QVT relations. In *ASE'09*, pages 3–14. IEEE CS, 2009. See also `http://www.modeltransformation.net/`. Last accessed: Nov. 2010.

A Appendix

A.1 Compilation of meta-models into colour sets

Listing 1 shows the procedure to compile meta-models with and without inheritance into colour sets. We assume that we can access the name of meta-models, classes and associations through the dot notation "$.\langle\text{name}\rangle$", the name of the variable associated to the meta-model M in a QVT-R specification S through "$\text{M}.\langle\text{id}\rangle$", the ancestors of a class through the dot notation "$.\langle\text{anc}\rangle$", and assume binary associations with named roles at both ends. If a role is missing, we take as role the name of the class in the association end. We assume that the two roles of an association have different names.

In the compilation, step 1 creates a page for the meta-model, step 2 creates a record and a place of the record sort for each class, and step 3 does the same for each association. Records associated to classes include an object identifier and one field for each attribute in the class or its ancestors. Records derived from associations store

```
Given a meta-model M and a QVT spec. S:
 1 Create a page called M.⟨name⟩+M.⟨id⟩.
 2 ∀ Class c ∈ M:
   2.1 Create a record r called c.⟨name⟩.
   2.2 Add a field $ID$ of sort $INT$ to r.
   2.3 ∀ Attribute a ∈ {c ∪ c.⟨anc⟩}:
      2.3.1 Add a field a.⟨name⟩ of sort a.⟨type⟩ to r.
   2.4 Create a Place p called as r of sort r.
   2.5 Create a Fusion Set called M.⟨id⟩+c.⟨name⟩ and
       include p in it.
 3 ∀ Association n with roles $r_1$ and $r_2$ ∈ M:
   3.1 Create a record r called $r_1$.⟨type⟩+``_''+
       $r_1$.⟨name⟩+``_''+$r_2$.⟨name⟩+``_''+$r_2$.⟨type⟩.
   3.2 Add two fields $r_1$.⟨name⟩ and $r_2$.⟨name⟩ of sort $INT$
       to r.
   3.3 ∀ Attribute a ∈ n:
      3.3.1 Add a field a.⟨name⟩ of sort a.⟨type⟩ to r.
   3.4 Create a Place p called as r of sort r.
   3.5 Create a Fusion Set called M.⟨id⟩+r.⟨name⟩ and
       include p in it.
```

Listing 1: Compiling a meta-model into CPN colour sets.

the identifier of the objects in each association end. The name of the records for the associations is given by the concatenation of names of the source and target types and roles.

A.2 Compilation of relations: Graphical pattern

The compilation of the relations in a QVT-R specification consists of two steps: the creation of colour sets and places to store the execution traces, and the generation of transitions modelling the relations. Listing 2 shows the pseudocode for the first step assuming relations without *when* and *where* clauses, which are handled in Section A.3 of this appendix. We assume that we can access the variable name of an object in a relation by using the dot notation "o.⟨id⟩", and all objects in both domains of a relation r through "r.⟨domains⟩". According to the standard, the traces store the identifier of all objects involved in a relation.

```
Given a QVT specification S:
 1 Create a page called ``Traces''.
 2 ∀ Relation r ∈ S:
   2.1 Create a record t called ``T''+r.⟨name⟩.
   2.2 ∀ Object o ∈ r.⟨domains⟩:
      2.2.1 Add a field o.⟨id⟩ of sort $INT$ to t.
   2.3 Create a record called ``LT''+r.⟨name⟩ of sort $list$
       ``T''+r.⟨name⟩.
   2.4 Create a variable called ``tr_''+r.⟨name⟩ of sort
       ``LT''+r.⟨name⟩.
   2.5 Create a Place p called ``Trace''+r.⟨name⟩ of sort
       ``LT''+r.⟨name⟩.
   2.6 Create a Fusion Set called ``Trace''+r.⟨name⟩ and
       add p in it.
```

Listing 2: Generating colour sets for the relation traces.

Listing 3 shows the translation of each relation into a CPN transition. The handling of identifiers for the created objects is managed through the use of a global counter called ID, which gets incremented each time the transitions fire (step 8). In particular, firing a transition increments the counter as many units as objects the relation creates. Then, if a relation creates several objects o_i, their identifier is assigned the value $ID - i$.

```
Given a QVT specification S:
∀ Relation r ∈ S:
 1 Create a page called r.⟨name⟩.
 2 Create a transition t called r.⟨name⟩.
 3 ∀ Domain d ∈ r.⟨domains⟩:
   3.1 ∀ Object o ∈ Domain d:
       3.1.1 Let P(o)={p: Place | p is called o.⟨type⟩ and
             belongs to the fusion set d.⟨id⟩+o.⟨type⟩} be
             the set of places associated to Object o.
       3.1.2 if Domain d is enforced, include in P(o) a
             place for each supertype $st$ of o.⟨type⟩, with
             name $st$ and fusion set d.⟨id⟩+$st$.
       3.1.3 Create all places in the set P(o).
       3.1.4 Create a variable named o.⟨id⟩ of sort $INT$.
       3.1.5 Add an arc from t to each p ∈ P(o), labelled
             with the variables and conditions in r. The
             field $ID$ is read from variable o.⟨id⟩ if
             the domain is checkonly, or from the global
             counter $ID$ if the domain is enforced.
       3.1.6 If the domain is checkonly, add an arc from
             each p ∈ P(o) to t. Label such arc with the
             variables and conditions in r.
 4 Create a place q named ``Trace''+r.⟨name⟩ of sort
   ``LT''+r.⟨name⟩.
 5 Add an arc from q to t, labelled ``tr_''+r.⟨name⟩.
 6 Add an arc from t to q, labelled with the
   concatenation of ``tr_''+r.⟨name⟩ and a record with
   the identifier of all objects in r (the field $ID$ of
   each object is obtained as in step 3.1.5).
 7 Add a guard to the transition that checks that
   the selected binding of objects is not in the list
   ``tr_''+r.⟨name⟩.
 8 Add an action to the transition that increments
   the global counter $ID$ as many units as objects the
   relation creates.
```

Listing 3: Compiling the relations into CPNs.

A.3 Compilation of relations: When and where

In order to handle the *when* and *where* sections of a relation, we have to perform the following modifications to Listing 3:

- For each relation r invoked in the *where* clause, we create an additional place and colour set called "Param"+r.⟨name⟩ to store the identifier of the objects used in the invocation. The calling relation is connected to the place with a write arc. The called relation r is connected to the place with a self-loop arc.
- For each relation r inspected in the *when* clause, we add a self-loop arc from the place "Trace"+r.⟨name⟩ of the inspected relation to the transition of the actual relation.
- If a relation is invoked from a *where* clause from which it receives certain parameter objects, then the arcs that write tokens to the places for these objects are not created (i.e. step 3.1.5 in Listing 3 is not performed for the received objects).
- If a relation checks certain object in the *when* clause, then the arc that writes tokens to the place for this

object is not generated (i.e. step 3.1.5 in Listing 3 is not performed for the checked objects).

In our current compilation, we restrict to *when* and *where* sections that contain sequences of calls to other relations, but not arbitrary formulae involving relations.

A.4 Model matching scenario

The procedure for generating transitions in the model matching scenario is similar to Listing 3, but both domains are checkonly, and hence places of both domains are connected to the transition with self-loop arcs. Moreover, we do not make use of the global counter ID, so that transitions are not added any action (i.e. step 8 is not performed) and arcs always use the variables used in the relation (i.e. in steps 3.1.5 and 6, the object identifiers are read from the corresponding variable).