Formal Support for QVT-Relations with Coloured Petri Nets

Juan de Lara[1] and Esther Guerra[2]

[1] Universidad Autónoma de Madrid (Spain), `jdelara@uam.es`
[2] Universidad Carlos III de Madrid (Spain), `eguerra@inf.uc3m.es`

Abstract. QVT is the OMG standard language for specifying model-to-model transformations in MDA. Even though it plays a crucial role in model driven development, there are scarce tools supporting the execution of its sublanguage QVT-Relations, and none for its analysis or verification. In order to alleviate this situation, this paper provides a formal semantics for QVT-Relations through its compilation into Coloured Petri nets, enabling the execution and validation of QVT specifications. The theory of Petri nets provides useful techniques to analyse transformations (e.g. reachability, model-checking, boundedness and invariants) and to determine their confluence and termination given a starting model. We also report on using CPNTools for the execution, debugging, and analysis of transformations, and on a tool chain to transform QVT-Relations specifications into the input format of CPNTools.

1   Introduction

Model-to-model transformation consists in translating a model from a source to a target language. This process is at the core of Model-Driven Engineering (MDE), where models are used to generate code, test, document and verify the applications to be built. Among the existing model-to-model transformation languages, QVT [15] stands out for being the transformation standard proposed by the OMG in the framework of the Model-Driven Architecture (MDA). QVT 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 [12, 13], and even less its verification or validation. This fact hinders its use in industry, where the complexity of models and transformations makes essential the development of tools and techniques for transformation analysis, which can be only built on the basis of a formal semantics for the transformation language. At present, QVT-R is given a semantics in terms of QVT-C, whose semantics is in its turn semi-formally defined. Thus, the MDE community would benefit from a clean, formal semantics for QVT-R enabling the analysis of transformations and serving as a reference for tool builders.

In this sense, Coloured Petri nets (CP-nets or CPNs in short) [8, 9] 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, transition persistence (i.e. conflicts) or reachability [8]. Many of these properties rely on the occurrence graph, a representation of the state space that can be model checked and used to determine termination and confluence of a transformation relative to a starting model. The CP-nets community has developed a number of tools – CPNTools [9] being the best known one – with a level of maturity that makes them usable for industrial projects.

In the present work, we profit from the theory and tools developed for this formalism by providing a formal semantics for QVT-R in terms of CPNs. This 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 and analysis of QVT transformations, overcoming the lack of support for QVT-R. The explicit and visual nature of CPN models allows debugging and validating the transformation execution graphically, while their executable semantics may serve as a reference implementation for tool builders. A prototype tool chain, based on the QVT-R parser of MediniQVT [12] and the code generation facilities of JET, 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.
Paper organization. Sections 2 and 3 introduce QVT-R and CPNs. Section 4 shows the compilation from QVT-R into CPNs. Section 5 presents our supporting architecture. Section 6 illustrates the use of CPNs for verification and validation of transformations. Section 7 compares with related research and Section 8 ends with the conclusions and lines for future work.

2   QVT-Relations

QVT-R is the highest-level of abstraction language of the QVT OMG standard [15]. It has a declarative nature and a dual graphical and textual syntax. In this language, a model-to-model transformation is made of relations with two or more domains (usually two). Domains are described by patterns similar to object diagrams. When a domain is marked as *enforced*, the models to which it is applied may be modified in order to satisfy the relation; whereas if it is *checkonly*, they are just inspected to check for disagreements.

Relations may contain *when* and *where* clauses. The former express conditions under which the relation needs to hold. They usually refer to other relations, to which they pass a number of parameters that appear as variables in the current relation. *Where* clauses may call other relations, similar to function calls in traditional programming. In addition, relations may be *top* or *non-top* level. The execution of a transformation requires that all its top-level relations hold,

whereas the non-top level ones only need to hold when invoked from the *where* section of other relations.

QVT-R uses the check-before-enforce (CBE) semantics [15]. Thus, before creating new objects, it is checked whether existing ones satisfying the constraints of the relation can be reused. Transformations may declare *keys* as unique identifiers for objects. These are used by the CBE semantics to decide whether to create a new object. The semantics of QVT-R is given by its compilation into QVT-C, relying on the synthesis of tracing mechanisms.

For illustrative purposes, we provide a simple transformation from a tiny subset of UML class diagrams into relational database schemas. The example is a simplification of the one given in the QVT standard [15]. The meta-models for the source and target languages are shown in Fig. 1.
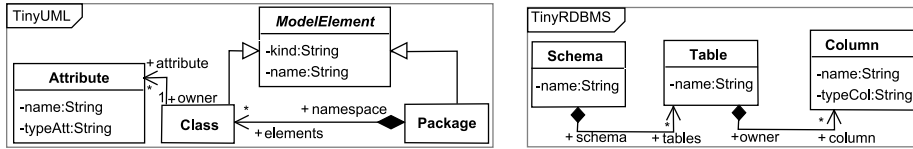

Fig. 1. The source and target meta-models.

The QVT-R transformation is shown next. It defines three relations, two of them being top-level, and an auxiliary function. All relations enforce the RDBMS domain. The first one checks that for each persistent package in a UML model, there is a schema with same name (given by the **pn** String variable). The second one states that for each persistent class there must be a table with same name. The *when* section specifies that this relation has to hold only if relation *PackageToSchema* holds for the package and schema to which the class and table belong. The *where* clause asks the *AttributeToColumn* relation to hold for the class, table and an empty prefix. Finally, the last relation requires that for each attribute of a class there is a column with a name made of the received prefix plus the class name. The type of the column is calculated by the auxiliary function *PrimitiveTypeToSqlType*.

```
transformation umlToRdbms(uml:TinyUML, rdbms:TinyRDBMS) {
 top relation PackageToSchema { // maps each package to a schema
   pn: String;
   checkonly domain uml p:Package {name=pn, kind='persistent'};
   enforce domain rdbms s:Schema {name=pn};
 }
 top relation ClassToTable { // maps each persistent class to a table
   cn, prefix: String;
   checkonly domain uml c:Class {namespace=p:Package {},
                                 kind='persistent', name=cn};
   enforce domain rdbms t:Table {schema=s:Schema {}, name=cn};
   when { PackageToSchema(p, s); }
```

```
     where { prefix=''; AttributeToColumn(c, t, prefix); }
}
relation AttributeToColumn { // maps an attribute to a column
  an, pn, cn, sqltype: String;
  checkonly domain uml c:Class {attribute=a:Attribute {name=an, typeAtt=pn}};
  enforce domain rdbms t:Table {column=cl:Column {name=cn, typeCol=sqltype}};
  primitive domain prefix:String;
  where { cn = if (prefix = '') then an else prefix+'_'+an endif;
          sqltype = PrimitiveTypeToSqlType(pn); }
}
query PrimitiveTypeToSqlType(primitiveType:String):String {
  if (primitiveType='INTEGER') then 'NUMBER'
  else if (primitiveType='BOOLEAN') then 'BOOLEAN' else 'VARCHAR' endif
  endif;}
}
```

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 [8, 9] for more
details. A CPN model can be seen as a bipartite graph made of two kinds of
nodes: *places* and *transitions*. The former represent the states of the net and
are depicted as ovals with the name inside. Transitions model actions and are
depicted as labelled rectangles. Places can be connected to transitions, and tran-
sitions to places, by means of *arcs*. As an example, Fig. 2 shows to the left a
CPN with three places and one transition (exported from CPNTools). The net
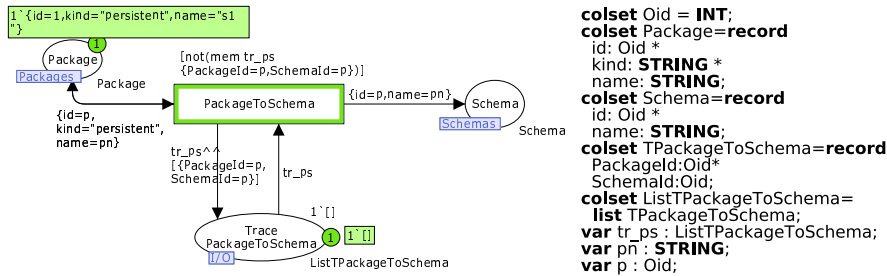actually models the relation *PackageToSchema* of the example transformation.

Fig. 2. Example CPN model: net (left) and colour set declarations (right).

Each place has a data type defining the kind of data it can contain, whose
name is usually depicted next to the place. Data types (called *colour sets*) are
declared in a language based on Standard ML, called CPN-ML [9]. The lan-
guage allows declaring simple colour sets – like unit, booleans, integers, strings

4

and enumerated – and compound ones, like products, records, lists, unions and subsets. The declarations for the example appear to the right of Fig. 2.

The state of a CPN is called its *marking*, and consists of a number of tokens located in the places. Each token contains data according to the colour set of the place where it is located. Places contain multi-sets of tokens. For the example, the *Package* and the *TracePackageToSchema* places contain one token each. The former is a record with value {id=1, kind="persistent", name="s1"}, while the latter is the empty list []. 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. An incoming arc to a transition indicates that the transition, if fired, will remove tokens from the connected places. Similarly, an outgoing place from a transition indicates that firing the transition will put tokens into the place. The tokens to be added or removed are given by the *arc expressions*. Transitions have a guard, shown between brackets, which is a boolean expression made of variables typed on the colour sets. The guard in the example checks the membership of a record in the tr_ps list.

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, i.e. if the incoming places have enough tokens to bind the variables appearing in the incoming arcs, the variables in the guard are bound, and the expression evaluates to true. In the example, transition *PackageToSchema* is enabled (and hence highlighted) because the arc from *Package* demands one token with value "persistent" in the field kind (which exists), while the guard demands a token containing a list without any record whose two fields are equal to p. Hence, the transition is enabled with the binding $b_1 = \langle p = 1, pn = \text{"s1"}, tr\_ps = [] \rangle$.

An enabled *step* is a finite, non-empty multi-set of bindings enabling certain transitions. An enabled step can *occur*, changing the marking of the enabled transitions by the multi-set. In this way, 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: (i) removes one token from *Package*, but then creates a token with same data there; (ii) adds one token to *Schema* with same name and id as the token from *Package*; and (iii) removes the token from the trace, but puts it back adding a new record at the end of the list. Note that firing the transition prevents it from firing again for the same binding, as the list in the trace place would contain a record making the guard expression false.

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 contains which is a graph representation of the reachable markings [8]. Section 6 will use some CPN analysis techniques to verify QVT-R transformations.

CPNTools offers additional hierarchical and modular modelling capabilities. A large net can be divided into *pages* that can be connected by means of *Fusion Places* and *Substitution Transitions*. The latter are transitions that stand for a

whole page of the net structure. For example, the net in Fig. 2 is represented as a single transition in Fig. 6, where we only show the *interface* places (marked *In*, *Out*, or *I/O* in the subnets). A fusion place is a place that has been 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*. For example, the place *Package* in Fig. 2 is the same as the place *Package* in Fig. 3, and both belong to the fusion set "Packages". Thus, these two 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

In this section we describe the compilation of QVT-R specifications into CPNs. We use the modular capabilities of CPNTools to create: two pages with places to store the objects of the source and target models; one page for each relation in the transformation; and a high-level view of the transformation with one substitution transition for each relation (linked to the corresponding page) and places depicting the *when* and *where* dependencies between relations.

4.1 Compiling the Meta-Models and the Initial Model

The first step is to compile the source and target meta-models into colour sets declarations. For this purpose, we calculate the transitive closure of the inheritance relations in the meta-models so as to copy the attributes and relations from parent to children classes. Then a *record* is generated 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 in case of an associative class.

As an example, the declarations for classes *Package* and *Schema* were shown in Fig. 2. As we will see in next subsections, further definitions will be added to store the traces of the relations, and parameter passing.

Next, we create one place for each created record, and populate it with tokens representing the model to be transformed. These tokens hold the values of each object attribute. 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. Fig. 3 depicts a TinyUML model to be transformed, and the corresponding generated places and initial marking. The model contains two classes with equal name (since the meta-model allows this), having one attribute each with equal name.

4.2 Compiling the Relations

Next, we compile the relations. We restrict to the case with one domain enforced and the other checkonly, and neglect CBE semantics and keys for the moment.
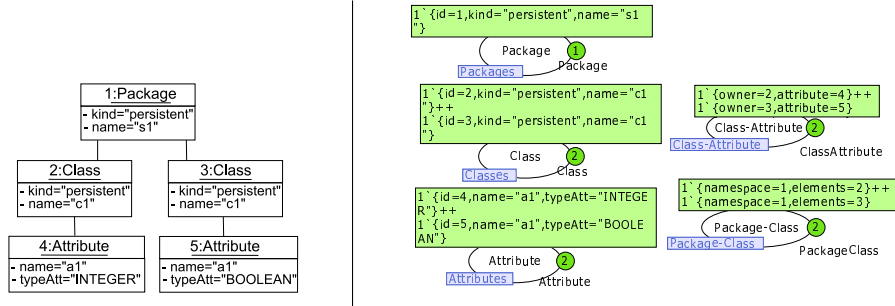
Fig. 3. Source model (left). Generated places and initial marking (right).

For each relation, we create a transition with its name in a different page. For each element in the relations domain, we create a place in 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, the transition is connected to the place. The arc inscriptions contain variables with same name as in the QVT relation, binding the different fields of the record. In checkonly domains we make the following simplification: 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`.

Finally, for each relation we generate a colour set for its trace which contains the identifiers of all objects appearing in the relation. This conforms to the standard semantics of the compilation of QVT-R into QVT-C [15]. Moreover, for each relation, we create one place with type equal to the list of traces of the relation. The transition inspects this place in order to check that the identifiers of the objects in the relation are not in the list. This avoids enforcing a relation more than once for the same binding. When the transition fires, the list of traces is added a new element with the processed objects. Later, we will also use the trace for the translation of the *when* clause.

Fig. 2 showed the transition generated for relation *PackageToSchema*. The trace place contains one token with the empty list. The read arc takes such list, the guard checks that a record with the identifiers of the involved objects is not present, and the write arc adds the record to the list when the transition fires. For simplicity, the created schema object is given the same identifier as the package, but in our implementation an ML function calculates unique identifiers.

4.3 Compiling the Where and When Clauses

After generating one transition per relation, we process the *when* and *where* clauses. The latter usually includes calls to other relations using as parameters bound objects of the current relation. For this reason, we create a colour set with fields corresponding to the parameters, and create a place with that type in a new fusion set. We add an arc from the transition to the place that 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, together with self-loop arcs.

As an example, Fig. 4 shows the transition generated for relation *ClassToTable*. The *ParamAttrCol* place is used to pass the three parameters to the relation *AttributeToColumn*. The marking shows the situation after firing the transition once, which creates a table with name `c1`. The transition remains enabled because there is a class which has not been processed, so its firing creates a new table also with name `c1`. As this does not conform to the CBE semantics, we will describe the needed modifications to the net in Section 4.5.
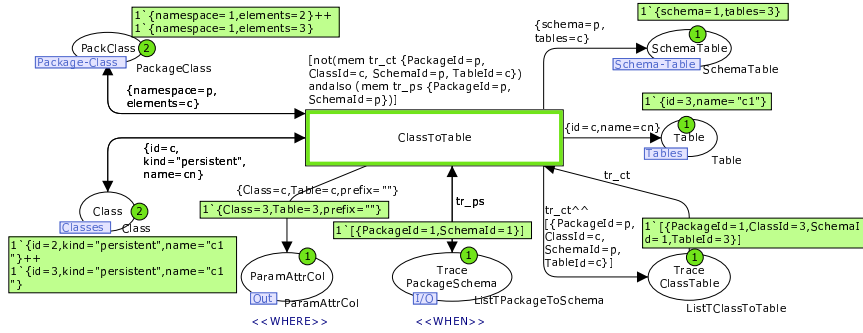

Fig. 4. Generated transition from relation *ClassToTable*.

The transition for relation *AttributeToColumn* is shown in Fig. 5, where the parameters are received from place *ParamAttrCol*. The parameters are not deleted from the place as, in general, a relation may need to be enforced more than once. Should a relation be called with different parameter sets (e.g. 2 parameters instead of 3), we will have to replicate the transition for each set.
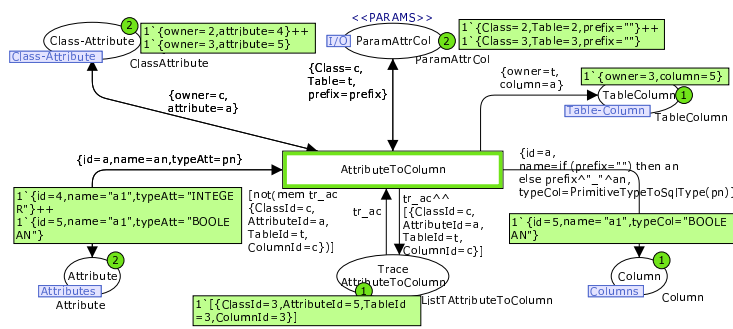

Fig. 5. Generated transition from relation *AttributeToColumn*.

8

The *when* clause is handled by querying the trace places. In particular, for each relation r_w appearing in the *when* clause of a relation r, a self-loop arc reading a token from the trace place of r_w is attached to the transition of r. Then, a guard is added to the transition of r demanding the read token to have a record with the values given by the actual parameters. Moreover, all arcs adding tokens to a place of an enforced domain which corresponds to an element passed as parameter are deleted. As an example, the transition generated from *ClassToTable* in Fig. 4 reads the list of traces from place *TracePackageSchema*, and the guard checks that the list contains a record indicating that the package and the schema have already been processed. Moreover, the place corresponding to the schema is not added tokens as the schema is a parameter in the *when* clause. In fact, with our simplified way of assigning object identifiers, we can make the following optimization: if an object is bound (i.e. present in some relation mentioned in a *when*, or comes as a parameter from a *where*), its attributes are not accessed, and the object is connected to some other one; then we do not use the place for the object, but the one for the link. This is why the places for the schema in Fig. 4 and the table in Fig. 5 are omitted.

4.4   Adding the High-Level View

Finally, we provide a high-level view of the transformation. This contains a substitution transition for each QVT relation, referring to the page with the relation details as described in previous sections. The view 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.

Fig. 6 shows the high-level view for the example. The top-level relations are shown with thicker border. Even though *ClassToTable* is top-level, it depends on *PackageToSchema* as the latter is referenced 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. The marking shows the result of the transformation, where two tables have been created. Since the result does not comply with the CBE semantics, we solve this problem in next section.
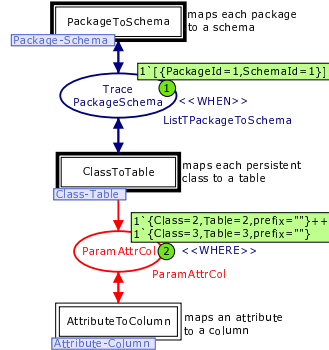

Fig. 6. High-level view.

4.5   Check-Before-Enforce (CBE) 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 created. The *keys* define when two objects are considered equal. The presented compilation has not taken this semantics into account. Even though traces avoided enforcing

a relation more than once for the same objects, we always created objects in the enforced domain instead of reusing them. Next we consider such semantics.

The idea is to generate several transitions for each relation. All transitions are mutually exclusive (at most one can fire at any given step), and each try 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 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 a list of the objects present in the model. In this way, testing if an object is not present amounts to ensuring that the corresponding record is not in the list.

Fig. 7 shows the two transitions generated from *PackageToSchema*. The left one creates a new schema if it is not found on the list `sch` taken from place *AllSchemas*, actually checked by the function `existsSchema` in the transition guard. The right one is executed if the schema exists and reuses the schema.
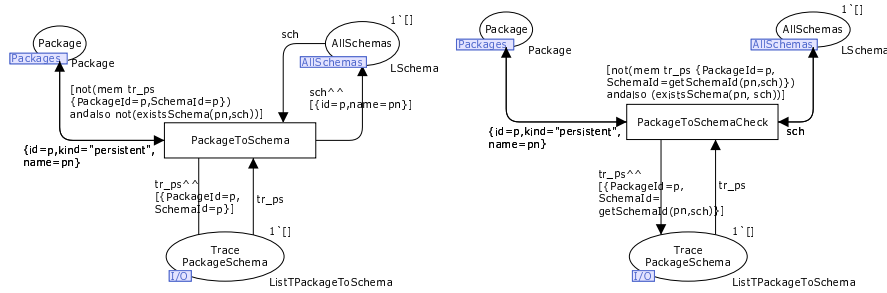

Fig. 7. Two transitions generated from *PackageToSchema* due to CBE.

The left of Fig. 8 shows the high-level view of the transformation with all transitions generated by the CBE semantics. The key for the table was its name and schema, and for the column its name and table. The marking shows some of the traces after executing the net, where only one table and one column are created, in conformance with the CBE semantics. The created model can be inspected in the page corresponding to the target meta-model.

5 Supporting Architecture

We have implemented a prototype to transform QVT-R specifications into the input format of CPNTools, for the moment without considering CBE semantics.
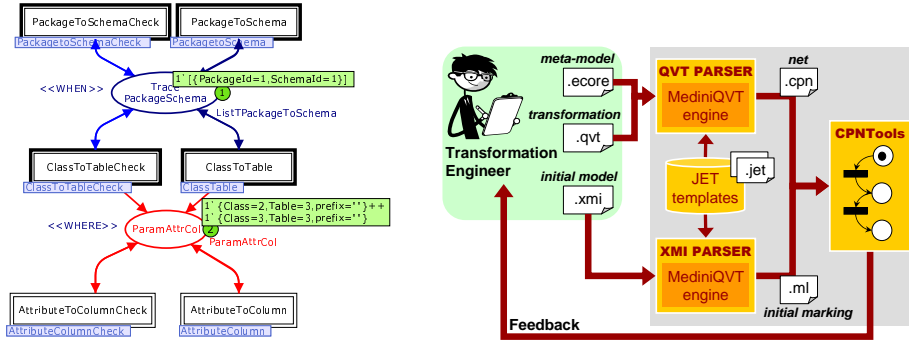
Fig. 8. High-level view with CBE semantics (left). Architecture of the solution (right).

The right of Fig. 8 shows its architecture. The engineer specifies the transformation using the textual format of QVT-R, and the source and target meta-models in *ecore*. We have built a code generator that parses these files using the MediniQVT parser [12], and then generates the input file to CPNTools through JET templates [10]. We also have developed another generator that, from an *xmi* model, generates a marking in a separate file that is read by the CPN model. In this way, no recompilation of the whole transformation is needed for different starting models. Then, the designer can execute the transformation using CPNTools, as well as to analyse it to find defects as we show in next section.

6 Verification and Validation of Transformations

This section presents some verification and validation techniques that are possible once the transformation is expressed in CPNs.

6.1 Verification

Many verification techniques for CPNs are based on the computation of the *occurrence graph* [8], a graph-based representation of the space of possible markings. Fig. 9 shows the graph for the example, considering CBE semantics and taking the starting model of Fig. 3. The graph shows the labels of two arrows, depicting the executed transition and part of the binding. To the right, the figure shows the TinyRDBMS models corresponding to nodes 13 and 14 of the graph. **Confluence**. A transformation is confluent if it yields a unique result for every possible starting model. We can investigate confluence by inspecting the terminal nodes of the occurrence graph. As we use lists, having more than one terminal node does not imply non-confluence: the lists may contain equal elements but ordered differently. Also, we obtain two different terminal nodes for models with the same structure, but different object identifiers. Our example however is non-confluent. The transformation creates one table (as both classes

have equal name) with one column. Processing the BOOLEAN attribute first creates a BOOLEAN column (nodes 16 and 13 in the graph), whereas processing the INTEGER first creates a NUMBER column (nodes 15 and 14). This is so because the key for attributes only considers their name and class but not its type. Considering also the column type solves this problem. Note however that CPNs only allows investigating confluence on individual starting models.

Another source of non-confluence is attribute computation using queries on enforced domains. For example, if the column name is computed as `cn=if (owner.column->size()=1) then '_'+ an else an;` we have non-determinism. This is so because the first column to be processed would be added a prefix '_', and this choice can be non-deterministic. Furthermore, if the table is added several columns, actually adding the prefix for the first column is wrong. Since QVT-R is declarative, the expression `cn=...` 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. Hence, if there are two columns no attribute should be added the prefix. Thus, "constructive" operational mechanisms would run into troubles. In our approach, we forbid attribute computations using queries on enforced domains.
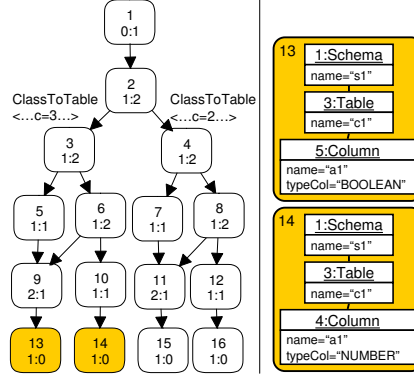

Fig. 9. Occurrence graph.

Termination. This is undecidable for graph and term rewriting systems [4]. QVT-R transformations can be non-terminating due to e.g. 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 the transformation always terminates for the given starting model. Our example transformation is terminating for the given initial model.

Relation Conflicts. Transition *persistence* allows discovering conflicts between relations. A transition is persistent if firing it does not disable other enabled transitions, and *weakly* persistent if it may disable itself at a different binding. Hence, if a transition is not persistent, it has conflicts and may lead to a non-confluent transformation. A conflict in QVT-R may arise if the execution of some relation depends on a query on an enforced domain, or if a relation A can be executed if some other B has not (by placing "*not B*(...);" in A's *when* section). In most cases all relations should be non-conflicting, leading to weakly persistent transitions in the CPN model. In the example, all transitions are weakly persistent as none disable others but may disable themselves. Persistence can be efficiently checked using the occurrence graph, and a sufficient condition for persistence exists by statically checking the underlying uncoloured net [14].

12

Boundedness and Invariants. A net is bounded if the number of tokens of all places remains bounded in all possible executions. This 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. Invariants are expressions on the marking that remain true in all reachable states. In a QVT-R transformation we find two types of invariants: the preservation of elements in checkonly domains, and maybe the non-creation of some type of element in enforced domains. Thus, we can check whether for a TinyUML model without attributes it is an invariant that no column is generated. These invariants are called synchronization invariants [5] and can be automatically computed.

Model Checking. Reachability analysis can be used to investigate whether some structure can be produced in the enforced domain, given an initial marking. This procedure can be automated, as CPNTools allows expressing properties to be checked on the occurrence graph by means of a CTL-like logic called ASK-CTL [3]. 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.

For instance, we can ask whether transforming our example model always produces a BOOLEAN column by using the command `eval_node INV(POS(NF( "Has Bool Column", hasColumn))) InitNode`, which returns false as we may obtain a NUMBER 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.

6.2 Validation with CPNTools

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. Similar to breakpoints in programming environments, one can set *monitors* establishing conditions (e.g. the marking exceeds a certain size, a transition occurs a certain 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 check if they are violated. The multi-view and hierarchical features permit visualizing the execution flow in the high-level page, and checking the created elements in the page corresponding to the meta-models.

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

7 Related Work

There are previous attempts to formalize QVT, such as the one in [7] for QVT-C. Regarding QVT-R, in [1, 11] the authors formalize it by using rewriting logic and Maude; however there is no comment about CBE semantics and no discussion on termination or confluence. In [6] the author uses OCL for representing the static semantics, and Alloy for the dynamics. Although Alloy permits execution and analysis, no discussion on analysis is given. That approach is similar to our previous work in [2], 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 meta-models and their 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. Other approaches like [16] use CPNs for transformations, but they have their own language, not QVT, and do not provide analysis techniques.

On the other hand, there are few tools for QVT-R. We can mention MediniQVT [12] and ModelMorf [13], but none of them provide analysis capabilities. Thus, we can see our work as a "low-cost" implementation of a QVT-R engine allowing both execution and analysis.

8 Conclusions and Future Work

In this paper we have presented an approach for the execution, verification and validation of QVT-R transformations through their compilation into CPNs. The approach supports when and where clauses and CBE semantics. 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 determine whether certain structures are created in enforced domains using model checking, invariants and boundedness analysis. Finally, we have demonstrated that CPN-Tools can be used for execution, verification and validation of transformations; and presented a tool that automates the code generation for it.

One limitation of our proposal is the full support for OCL, which would require a complex compilation into ML. Up to now we support a small subset enough to translate the auxiliary query of the example. Complex queries involving negation would require using tokens with lists also in checkonly domains. We are currently improving our tool chain and defining back-annotation mechanisms so that the user does not realise that the execution is based on CPNs, e.g. by translating the final marking into XMI. It would be also 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 [5]

and translating such optimizations into QVT, or the verification of properties independently of the marking. We also plan to complement our analysis techniques with the automatic generation of initial markings for the nets.

References

1. 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.
2. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. An invariant-based method for the analysis of declarative model-to-model transformations. In *MoDELS'08*, volume 5301 of *LNCS*, pages 37–52. Springer, 2008.
3. A. Cheng, S. Christensen, and K. H. Mortensen. Model checking coloured petri nets exploiting strongly connected components. In *WODES*, pages 169–177, 1996.
4. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer, 2006.
5. 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.
6. M. García. Formalization of QVT-Relations: OCL-based static semantics and Alloy-based validation. In *MDSD today*, pages 21–30. Shaker Verlag, 2008.
7. J. Greenyer. A study of model transformation technologies: Reconciling TGGs with QVT. Master's thesis, University of Paderborn, 2006.
8. K. Jensen. *Coloured Petri nets basic concepts, analysis methods and practical use (Monographs in theoretical computer science)*. Springer, 1997.
9. 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://wiki.daimi.au.dk/cpntools`.
10. JET. `http://www.eclipse.org/modeling/m2t/?project=jet`, 2009.
11. 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.
12. MediniQVT. `http://projects.ikv.de/qvt/`, 2009.
13. ModelMorf. `http://www.tcs-trddc.com/ModelMorf/index.htm`, 2009.
14. 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, 2004.
15. QVT. `http://www.omg.org/docs/ptc/05-11-01.pdf`, 2005.
16. M. Strommer and M. Wimmer. A framework for model transformation by-example: Concepts and tool support. In *TOOLS*, volume 11 of *LNBIP*, pages 372–391. Springer, 2008.