Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios

Technical Report
tr-ri-07-284

Ekkart Kindler and Robert Wagner
Department of Computer Science
University of Paderborn
D-33098 Paderborn, Germany
[kindler|wagner]@upb.de

June 2007

Abstract

Triple Graph Grammars (*TGGs*) are a technique for defining the correspondence between two different types of models in a declarative way. The power of TGGs comes from the fact that the relation between the two models cannot only be defined, but the definition can be made operational so that one model can be transformed into the other in either direction; even more, TGGs can be used to synchronize and to maintain the correspondence of the two models, even if both of them are changed independently of each other; i. e., TGGs work incrementally.

TGGs have been introduced more than 10 years ago by Andy Schürr. Since that time, there have been many different applications of TGGs for transforming models and for maintaining the correspondence between these models. To date, there have been several modifications, generalizations, extensions, and variations. Moreover, there are different approaches for implementing the actual transformations and synchronizations of models. In this paper, we present the essential concepts of TGGs, their spirit, their purpose, and their fields of application. We also discuss some of the extensions along with some of the inherent design decisions, as well as their benefits and caveats. All these are based on several year's of experience of using TGGs in different projects in different application areas.

Contents

1 Introduction **1**

2 Ideas and Principles **1**
2.1 Models and Meta-models 2
2.2 Graph Grammars 3
2.3 Triple Graph Grammars 6
2.4 Application Scenarios 15
2.5 Discussion 21

3 Advanced Concepts **23**
3.1 Attributes 23
3.2 Constraints 25
3.3 Reusable Nodes 26
3.4 Modes 28
3.5 Short hand Notations 28

4 Usability **30**
4.1 Specification by Example 30
4.2 Specification in Graphical Syntax 48

5 Realization **50**
5.1 Generative Approach 50
5.2 Interpreted Approach 54

6 Tool Support **60**
6.1 TGG-Compiler 60
6.2 TGG-Interpreter 61
6.3 Tool Adapter 63

7 Related Work **65**
7.1 Model Transformation 65
7.2 Model Integration 67
7.3 Model Synchronization 67
7.4 Usability 69

8 Conclusion and Future Work **70**

1 Introduction

Triple Graph Grammars (*TGGs*) have been introduced by Andy Schürr in 1994 as a technique for model transformation [40]. Their main advantage is that TGGs allow us to define a transformation in a declarative way and still execute the transformation in both directions. Actually, TGGs are even more powerful: They can be used to define the relation between two types of models and, then, to transform a model of one type into another, to compute the correspondence between two existing models, or to maintain the consistency between the two types of models as defined by the TGGs. When one of the models is changed, the other one can be change accordingly, which means that the transformations or synchronizations can be applied incrementally.

Over the years there have been many applications of TGGs for model transformation, and – based on the experience of these applications – there has been much research on TGGs and many modifications, generalizations, extensions, and variations of TGGs. Some of these modifications and extensions fit the idea and principles of TGGs quite naturally; others need a more careful investigation. Moreover, there are different implementations that actually execute the transformations and synchronizations defined by TGGs.

It turned out that the concepts of *QVT* (*Query/View/Transformation*) – the new OMG-standard for model transformation – are strikingly similar to the concepts of TGGs and at least the core of QVT can be easily mapped to TGGs [18].

Though there are several implementations of TGGs and many papers presenting the use of TGGs, there is no paper yet that gives a concise survey covering the idea and principles of TGGs, discusses their motivation from the application point of view, and weighs the benefits of different extensions and modifications. This is why we set out to write this paper.

2 Ideas and Principles

In this section, we explain the idea of *TGGs* and the underlying principles. The exact definition is a slight extension[1] of the original definition of Schürr [40], but still in accord with its original intension. Schürr formalized the semantics by a double-pushout approach; here we give a concise definition of the syntax and the semantics of TGGs. But, we do not dwell on their formalization in terms of category theory.

Actually, we present the approach not from the graph theoretic or category theoretic point of view, but from a software engineering point of view: We use the concept of a *typed graph* and the concept of a UML *object diagram*[2] interchangeably; where the types of nodes and arcs of a graph are defined by a UML *class diagram*, the *type of the graph*.

[1] For the experts, we present TGGs for typed graphs right away and correspondence nodes may have multiple connections to the source and target model.

[2] Object diagrams are also called *instance diagrams*.

2.1 Models and Meta-models

The models that are to be related and transformed by TGGs will be represented as object diagrams; and a class diagram represents the set of models to be considered. As an example of a set of models, we consider a simple version of Petri nets. Figure 1 shows a simple Petri net model. It consists of *places*, *transitions*, and *arcs*, where the places are graphically represented as circles, the transitions as squares, and the arcs as arrows. Moreover, some places contain a *token*, which is graphically represented by a black dot inside the corresponding place.

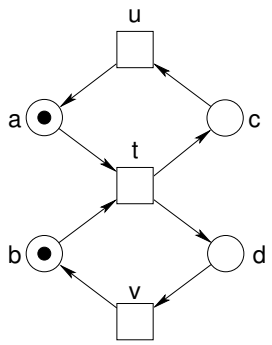

Figure 1: A Petri net

This structure of Petri nets is formalized in the UML class diagram of Fig. 2: A *Petri net* consists of *nodes* and *arcs*, where a node can be either a *transition* or a *place*; all these concepts are represented as *classes* in the class diagram. The class *Node* is abstract, since a concrete node needs to be a transition or a place. An arc connects two nodes, which is represented by the two *associations* between the classes *Node* and *Arc*. In Petri nets, it is not allowed to have an arc between two places or between two transitions; this condition is expressed by the OCL constraint for the class *Arc*[3]. The association between the class *Place* and the class *Token* indicates the tokens belonging to each place. Since the UML class diagram captures the concepts of all Petri nets models, such kind of UML diagrams are often called *meta-models*; a concrete Petri net, such as the one shown in Fig. 1 is a *model* and an instance of the meta-model.

Actually, Fig. 1 shows a Petri net in its graphical representation, which is often called its *concrete syntax*[4]. In UML, a Petri net can be represented as an object diagram. The object diagram corresponding to the Petri net of Fig. 1 is shown in Fig. 3. Of course, this is not very readable anymore, since arcs are now explicitly shown as *objects*. The type of each object is indicated by the name of the class following a colon. The relation to other objects is indicated by *links*. This kind of representation of a Petri net model is called a model in

[3]Note that the OCL constraints are not important for the TGG concepts presented later. They are only relevant for concisely capturing the legal Petri net models, which is not the point of TGGs. We left them in the UML model only for completeness sake.

[4]Note that the exact positions and dimensions of the places, transitions, and arcs are not captured in our meta-model for Petri nets. Of course, this information could be captured in the meta-model; we omitted this information here in order to have simpler meta-models.

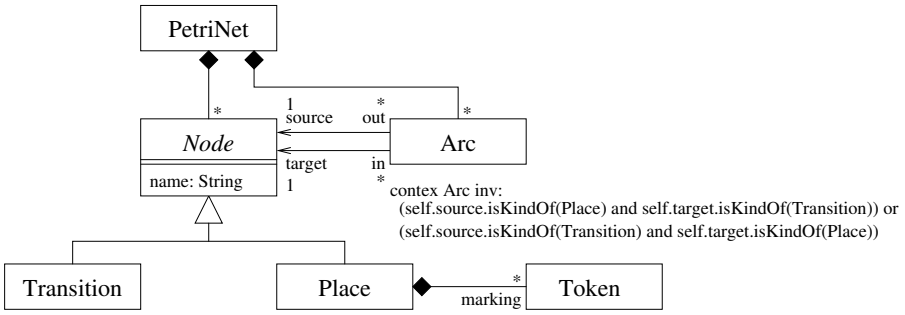

Figure 2: A class diagram: A meta-model for Petri nets

abstract syntax. And this will be the models on which transformations, and in particular our TGG-transformations, work. Clearly, object diagrams are some

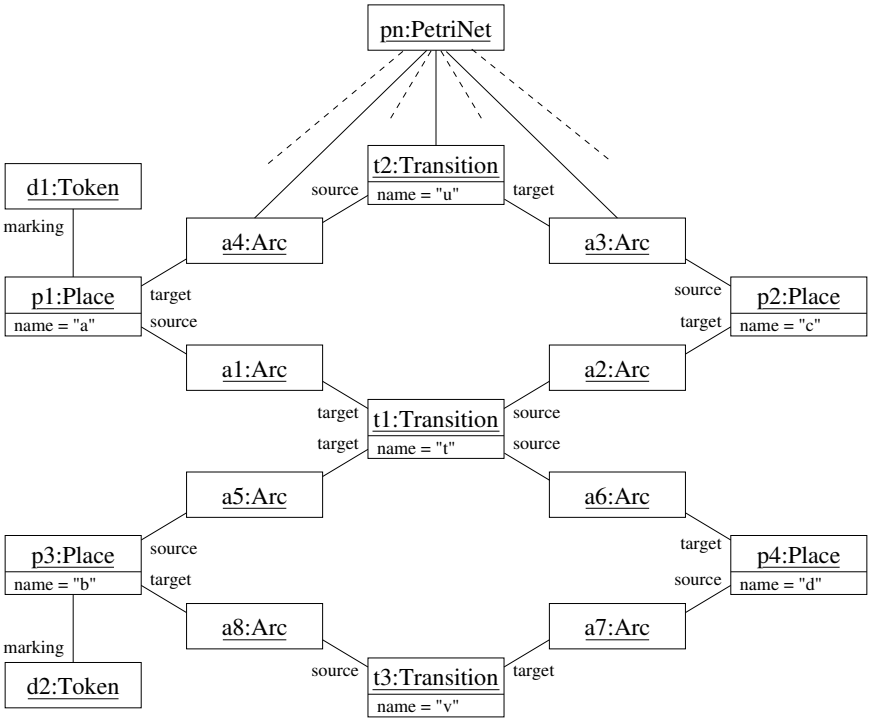

Figure 3: An object diagram: The Petri net in abstract syntax

version of typed graphs, which is the reason for applying techniques from graph grammars for model transformation.

2.2 Graph Grammars

In order to present the idea of TGGs, we need to discuss graph grammars first. But, we use a restricted form of graph grammars only: graph grammars without deletion rules. The graphs to be transformed will be object diagrams.

Figure 4 shows a simple graph grammar rule for our Petri net example. Basically, it consists of a pair of object diagrams. The first object diagram is

the left-hand side of the graph grammar rule. In our example, it consists of three objects: two places and one token with a link to the first place. These elements occur in the right-hand side of the graph grammar again, which is represented by the second object diagram. The use of the same names p, q and d, indicates which are the elements that occurred in the left-hand side already[5]. Additionally, there are a new transition and two new arcs, which connect the transition to the two places. Basically, the rule says that we can add a transition between two places, if the source place has a token.

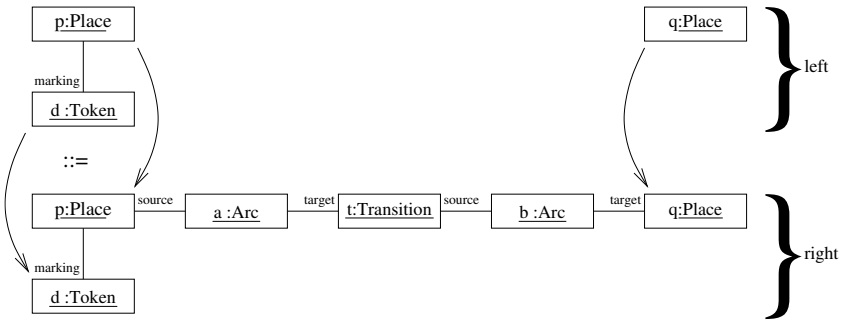

Figure 4: A graph grammar rule

The semantics of a graph grammar rule is similar to classical grammars in formal languages. A graph grammar rule can be applied to some graph, i. e., to an object diagram in our example. Here we apply the graph grammar rule from Fig. 4 to the object diagram from Fig. 3. In order to apply the rule at a particular position in this object diagram, we need to map the nodes and the links of the left-hand side of the rule to the objects and links of the object diagram. In our example, we map the node p of the graph grammar to object $p3$ of the object diagram, node d to object $d2$, and node q to object $p4$; the link between p and d in the rule is mapped to the link between $p3$ and $d2$ in the object diagram. Of course, the types of the objects must match and all links, which are in the left-hand side of the rule, must be there between the corresponding nodes in the object diagram. Then, this mapping is called matching left-hand side of the rule to the object diagram. Note that the given match is only one out of six other possible matchings. If we have found a matching mapping, we can *apply* the graph grammar rule in this mapping; this means that we insert new copies for all the objects and links which occur in the right-hand side of the rule, but not in left-hand side of the rule, where we keep the context of the mapping. In our example, this means that we introduce the transition $t4$ and the arcs $a9$ and $a10$ along with the corresponding links as shown in Fig. 5.

Applying a graph grammar rule changes an object diagram in a similar way the application of a string grammar rule changes a character string. In our example, we could apply this rule over and over again with different or even the same matchings, introducing more and more elements – though this would not make much sense.

[5]Moreover, we have drawn arrows between the corresponding elements, in order to emphasize this relation.

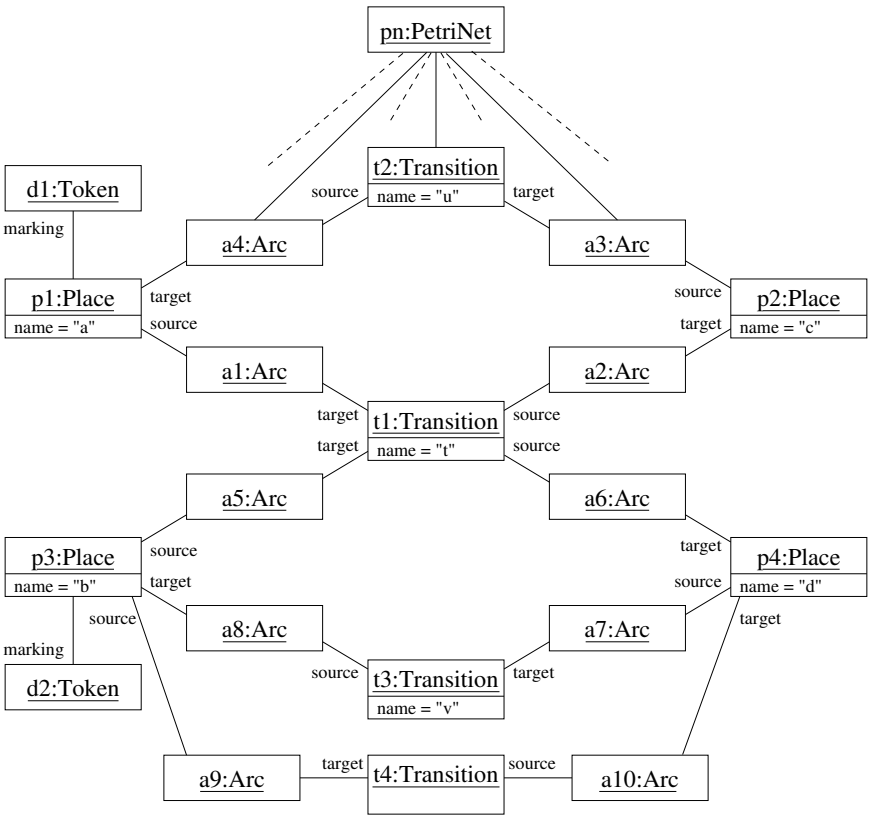

Figure 5: The Petri net after applying the rule

Note that in our graph grammar rule, the right-hand side contains all the elements that occurred in the left-hand side. This is called a *non-deleting* rule. In the context of our paper, all rules will be non-deleting rules. Non-deleting rules can be represented in a more concise way: Figure 6 shows the short hand form for the rule from Fig. 4. The black objects and links represent the elements that occur on both sides of the graph grammar rule; the green objects and links, which in addition are labeled with ++, represent the elements occurring on the right-hand side of the rule only. The labels ++ emphasize their meaning: these are the nodes to be *added* to the object diagram once the black nodes are matched in the original object diagram and the rule is applied.

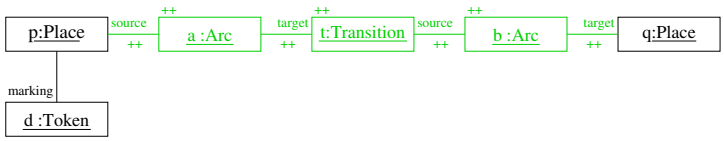

Figure 6: Short hand form for the graph grammar rule

Note that in general, graph grammar rules can refer to attribute values of objects. Since there are some subtleties concerning attributes, we ignore attribute values for the moment, and we will deal with attributes only later (see Sect. 3.1).

2.3 Triple Graph Grammars

Basically, graph grammars can be used for defining the dynamic evolution of a single model. Triple graph grammars allow us to define the relation between to different kinds of models and – as will be discussed later – to transform one kind of model into the other. Here, we will consider a simplified example from [28]: We transform a construction plan for some toy-train constructed from some standard components to a Petri net, which defines the dynamic behavior of the toy-train.

A construction plan for the toy-train consists of two different types of components: *tracks* and *switches*. These components have *ports* at which they can be mechanically connected. A track has a single *in-port* and a single *out-port*; whereas a switch has a single *in-port*, but two *out-ports* or vice versa two *in-ports* and a single *out-port*. Figure 7 shows a simple example of the graphical representation of such a construction plan, which we call a *project*. The different components are shown as boxes, the in-ports of a component are shown as small circles attached to these boxes, and the out-ports are shown as small squares. The connection of an in-port to an out-port is shown as an arrow from the in-port to the out-port. The meta-model for such a project is shown in Fig. 8. Note that, in this meta-model, we omitted the restriction of the number of ports for the particular type of component. We assume that the editor for such construction plans takes care of constructing only syntactically correct projects.

Next, we will define how such a construction plan and its different elements correspond to a Petri net. Figure 9 shows how each element of the construction

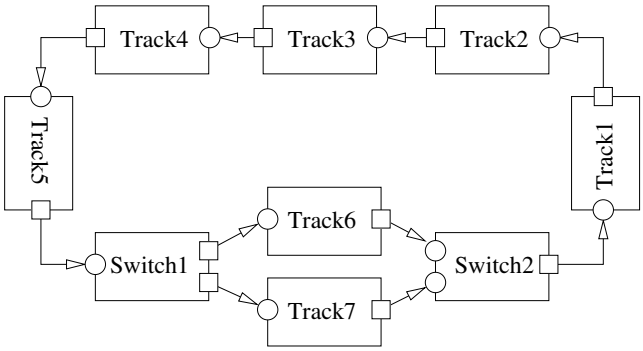

Figure 7: A construction plan for a toy train

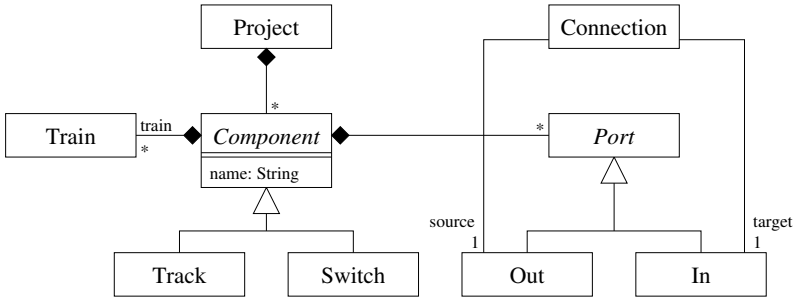

Figure 8: The meta-model for projects

plan should be mapped to a Petri net – informally in graphical syntax. Next we will define this relation formally by TGG-rules; these rules refer to the elements of both models, the construction plan and Petri nets, in abstract syntax, i. e., to object diagrams.

The first step toward a TGG-rule is from the graphical syntax to the abstract syntax. Figure 10 shows a track component and its corresponding Petri net in abstract syntax – the counterpart to the top left pair in Fig. 9.

Figure 11 shows the full TGG-rule for this relation: It defines how a *Track* component corresponds to a Petri net. On a first glance, this rule looks like a graph grammar rule. The only difference is that, now, there are three 'lanes' which define the different *domains*. On the left-hand side is the domain for the project, on the right-hand side is the domain for the Petri net, and in the middle is the part defining the correspondences between the elements of the different models (a meta-model for these elements is shown in Fig. 17); these objects are called *correspondence nodes*. The meaning of this TGG-rule is as follows: The black parts, which are not marked with ++, represent a *Project* which corresponds to a *PetriNet* via *Pr2PN*. The green parts, also marked with ++, insert a *Track* component along with its two ports to the project, as well as the corresponding *Place*, *Transition* and *Arc* into the corresponding *PetriNet*. This defines, how a *Track* component inserted to a project corresponds to inserting a Petri net fragment to the Petri net. The detailed correspondence nodes keep track of even more detailed correspondences between the ports of the project and the places and transitions of the Petri net.

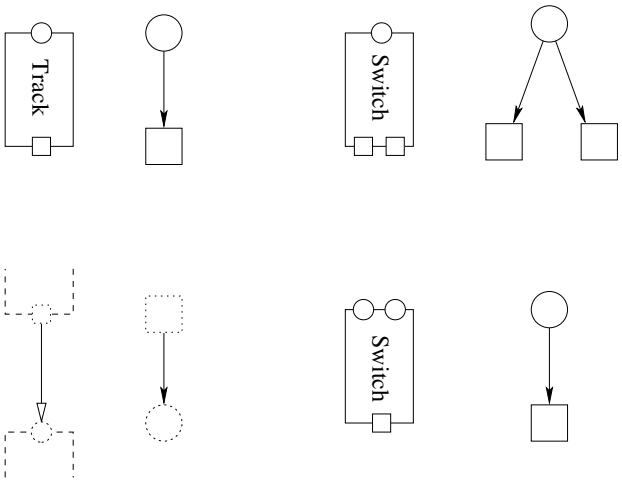

Figure 9: Informal mapping of project elements to Petri nets

Figure 10: Object diagram of a track and its Petri net

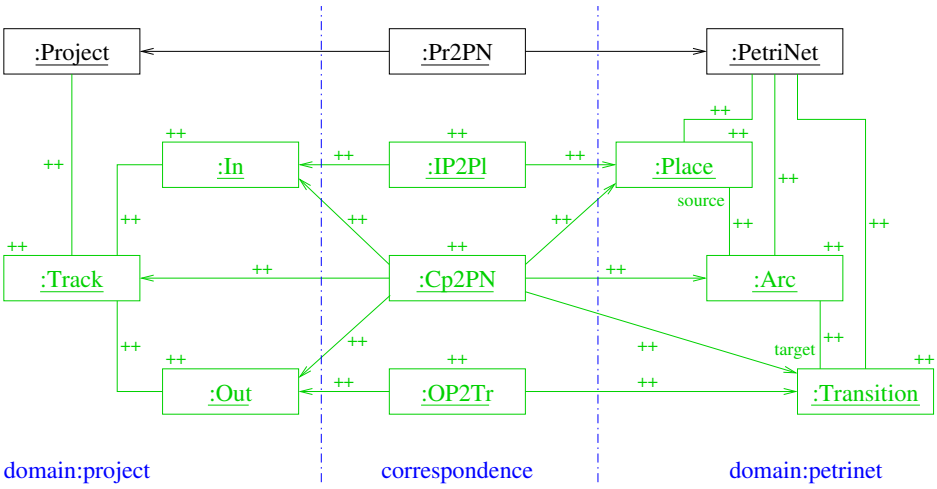

Figure 11: TGG-rule for component *Track*

This interpretation is in accord with the graph grammar interpretation. It says, how the project and the Petri net can be generated alongside. The only difference is that the TGG-rules are considered to generate both models alongside. This way, they define legal correspondences between models of different types.

Before discussing this in more detail, we present the TGG-rules for the remaining components and for connections. Figure 12 shows the TGG-rule for the switch that splits up a track into two. Again the in-port of the component corresponds to a place in the Petri net, and the two out-ports correspond to two transitions, and there is an arc from the single place to the two transitions (see Fig. 9 for the idea in graphical syntax). The rule for the switch which joins two tracks is shown in Fig. 13. It is similar, but there is only one place corresponding to both in-ports, which exactly reflects the join.

The TGG-rule for the connections, shown in Fig. 14, is different. When a connection is inserted between an out- and an in-port, which correspond to a transition and a place already, a corresponding arc will be inserted between this transition and place.

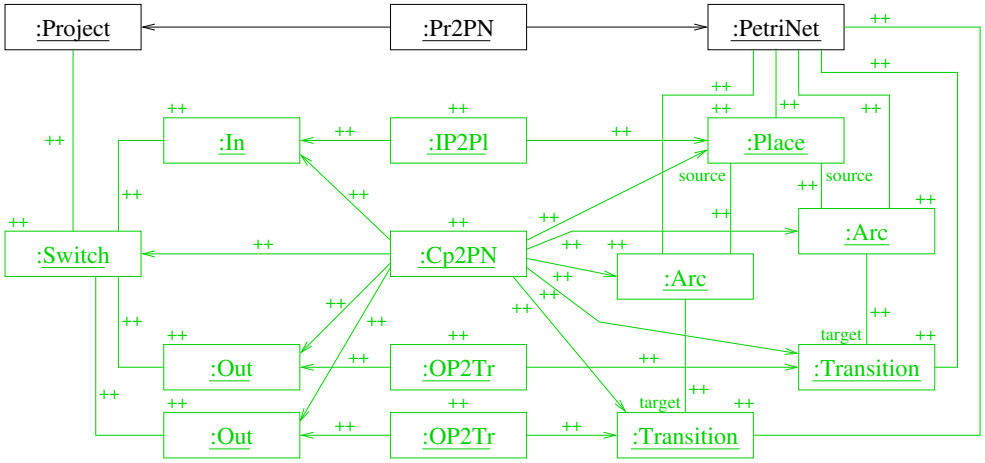

Figure 12: TGG-rule for a splitting *Switch*

At last, we introduce a rule that defines the correspondence between the trains that are on some components and the tokens in the Petri net. Each train on a component corresponds to a token on the corresponding place. This is represented by the TGG-rule shown in Fig. 15. In combination with the meta-model from Fig. 2, 8, and 17, these TGG-rules define how to construct corresponding pairs of projects and Petri nets, where we should start from the situation shown in Fig. 16, an empty project corresponding to an empty Petri net. This is called a TGG-*axiom*. Every pair of object diagrams for projects and Petri nets that can be constructed by applying the graph grammar rules, starting from this axiom at any matching position as long as we like, represents a legal relation between the two kinds of models. This is the semantics of a set of TGG-rules.

Figures 18–21 show how a simple project consisting of a sequence of three tracks along with its Petri net model can be constructed by applying the TGG-

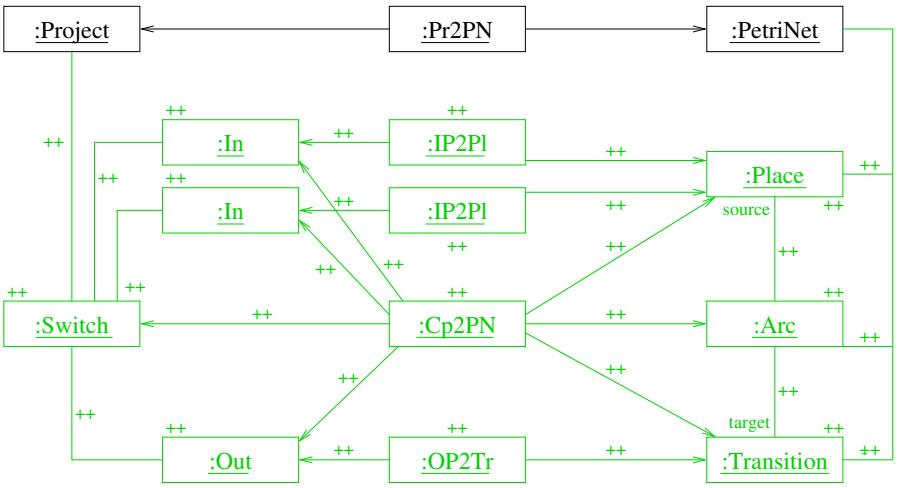

Figure 13: TGG-rule for a joining *Switch*

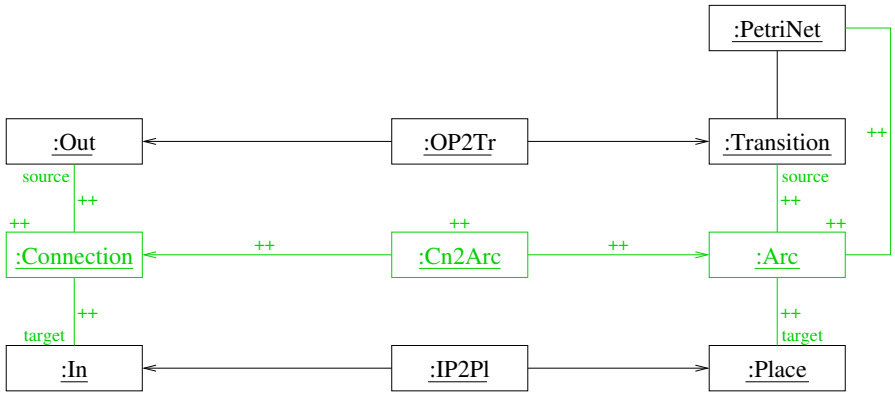

Figure 14: TGG-rule for a *Connection*

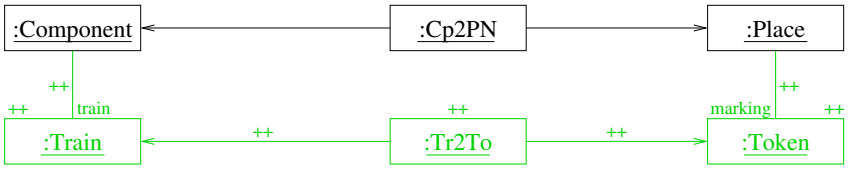

Figure 15: TGG-rule for a *Train*

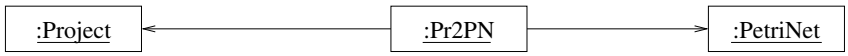

Figure 16: The axiom

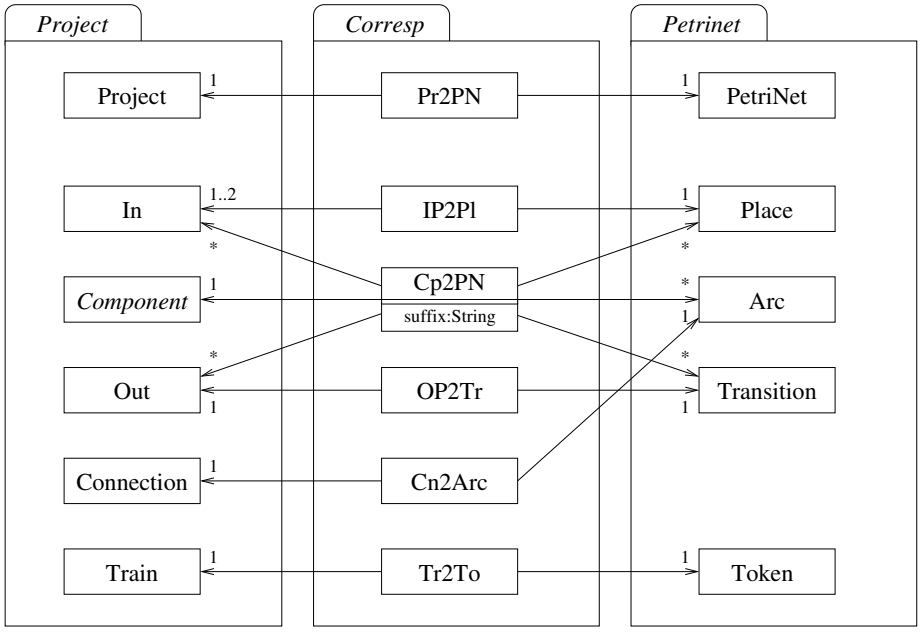

Figure 17: Meta-model for the correspondence objects

rule for tracks three times and the TGG-rule for connections twice. Figure 22 shows the corresponding models in graphical syntax.

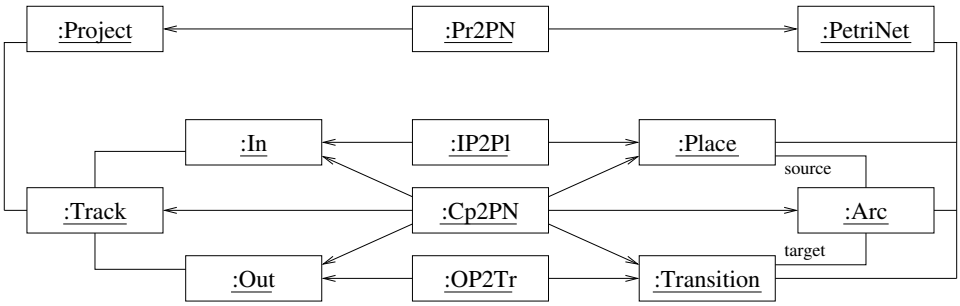

Figure 18: Applying the track rule to the axiom

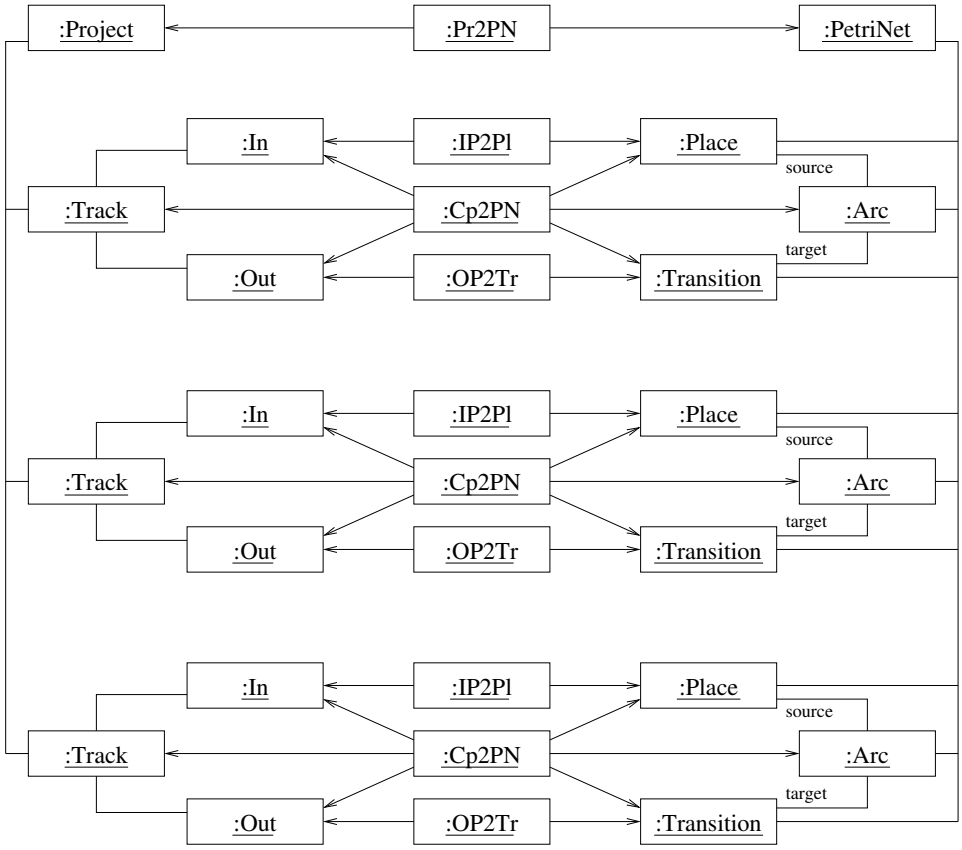

Figure 19: Applying the track rule two more times

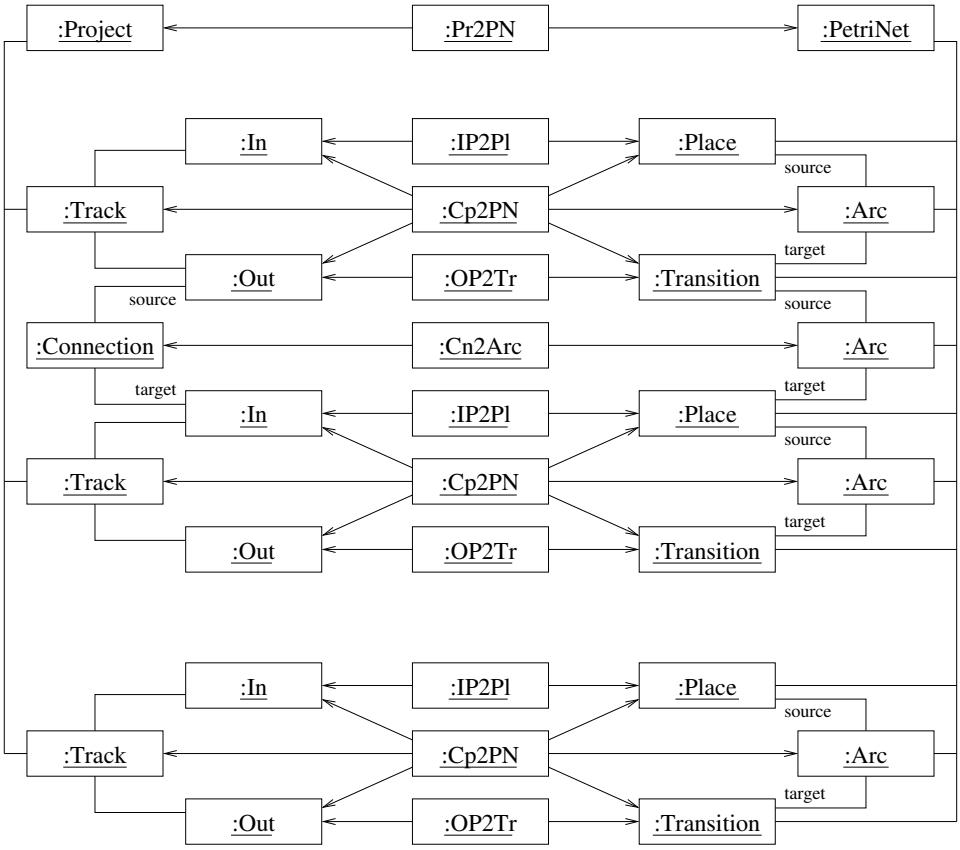

Figure 20: Applying the connection rule once

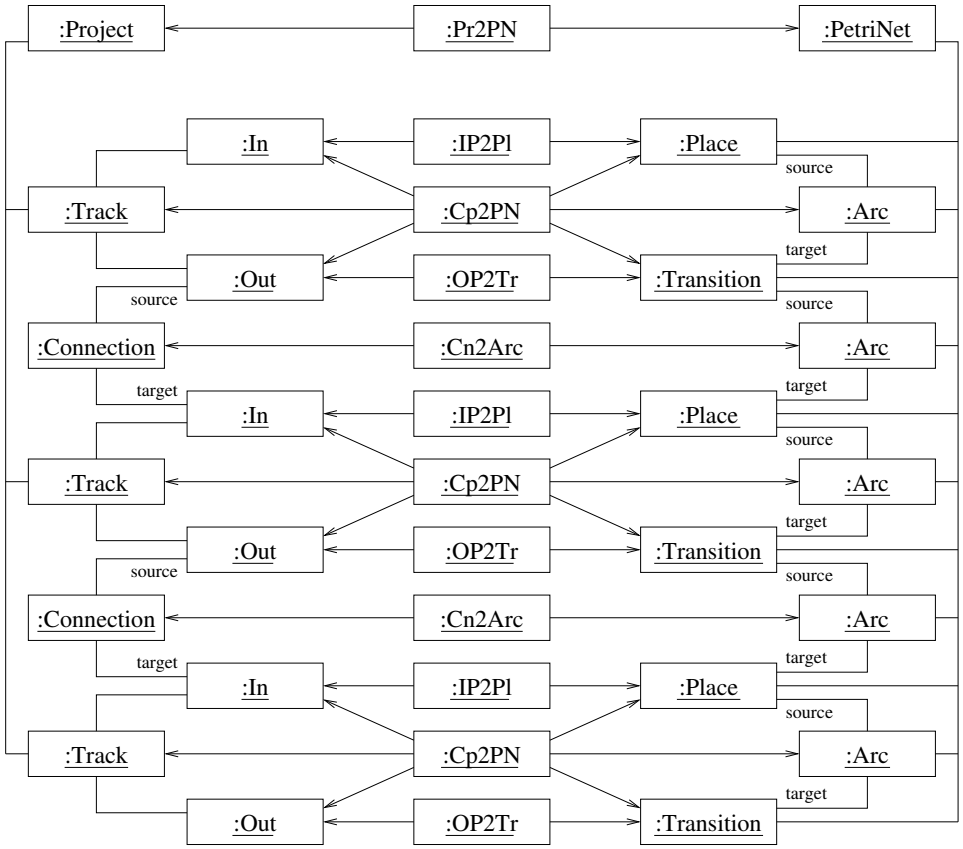

Figure 21: Applying the connection rule a second time

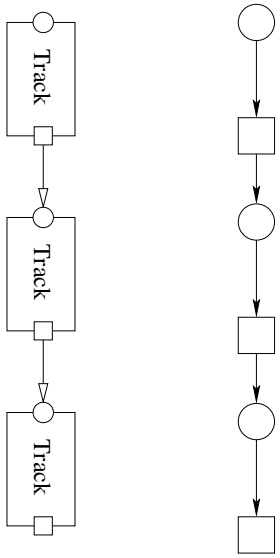

Figure 22: A pair of corresponding models in graphical syntax

2.4 Application Scenarios

In the previous section, we have had a look at the semantics of TGGs by generating the three parts, the source model, the target model, and the correspondences, in parallel – this way, we obtain pairs of models which are related to each other by the correspondence model. This definition is very much like the definition of the language of a classical string grammar by the words that are generated by it; the TGGs only generate pairs of related models instead of character sequences.

In practice, however, string grammars are seldom used for generating the words of a language; rather grammars are used for checking whether a given word is in the language or for parsing[6] a given character string into its structure: a syntax tree or derivation tree. Likewise, TGGs will be used for different purposes, which will be discussed below.

2.4.1 Model Transformation

The most obvious application scenario of TGGs is the transformation of one model into another: *model transformation*. In this scenario, one of the two models on either side exists already, and we would like to generate a corresponding model on the other side. For example, there could be a project and we would like to generate the corresponding Petri net. Since the project occurred on the left-hand side of our TGGs, this is typically called the source side, and the Petri net on the right-hand side is considered to be the target. Therefore, this transformation would be called a *forward transformation*.

In order to do this transformation, we start from the existing object diagram of a project and extend it as shown in Fig. 23. Then, we try to match the source domain of the rules of the TGG to the existing project model, and add the missing correspondence nodes and nodes of the Petri net of that rule to the correspondence model and to the Petri net. Figure 24 shows the situation after applying the TGG-rule for tracks twice to the situation of Fig. 23; Fig. 25 shows one additional application of the TGG-rule for connections. Once we have fully matched the project model, we have generated the corresponding Petri net model. For the situation shown in Fig. 23, we will eventually end up in the situation shown in Fig 21– which, again, corresponds to the pair of models shown in Fig. 22. The Petri net model on the right-hand side is the result of the transformation.

In principle, the transformation works also in the other direction, i. e., a Petri net could be transformed into a project. This is often called a *backward transformation*. Actually, TGGs are neutral with respect to the direction of the transformation, it only depends on which side is called the source or the target model. Sometimes, it is not even clear, what should be the source and what should be the target, so we simple call them *domains*. Then we must indicate which domain should be used as source and which as target for a transformation.

For this particular example, however, there are some problems with the transformation from a Petri net to a project: Firstly, there are Petri nets that do

[6] Actually, the grammar is used for constructing a parser.

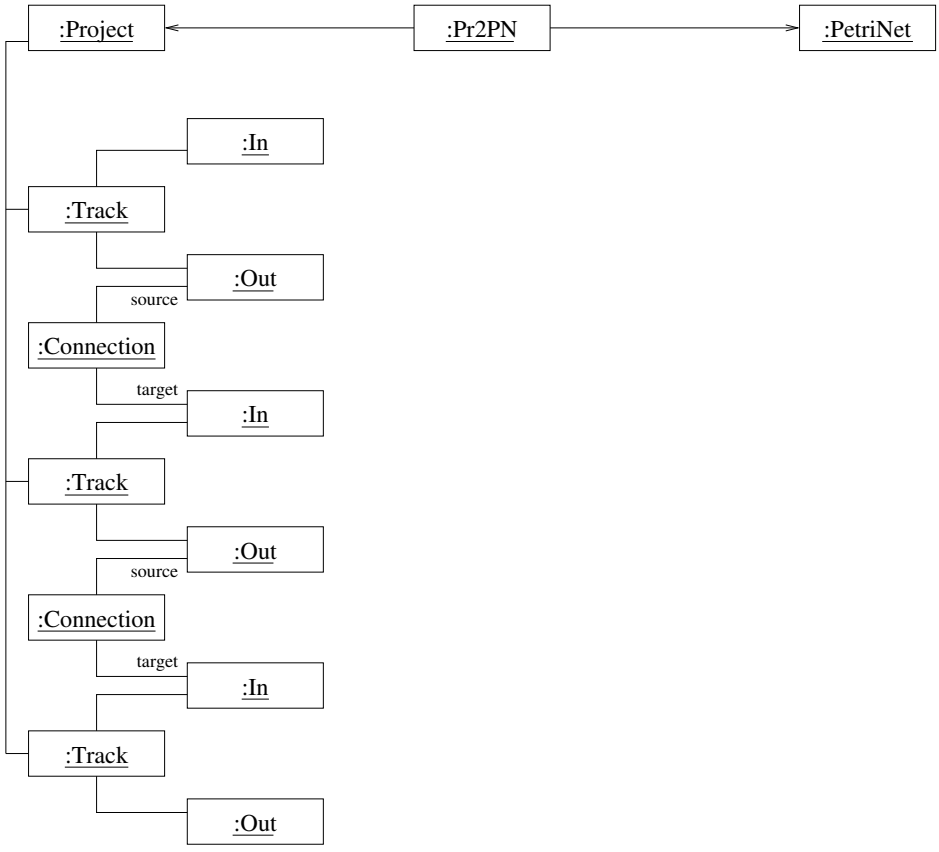

Figure 23: Forward transformation

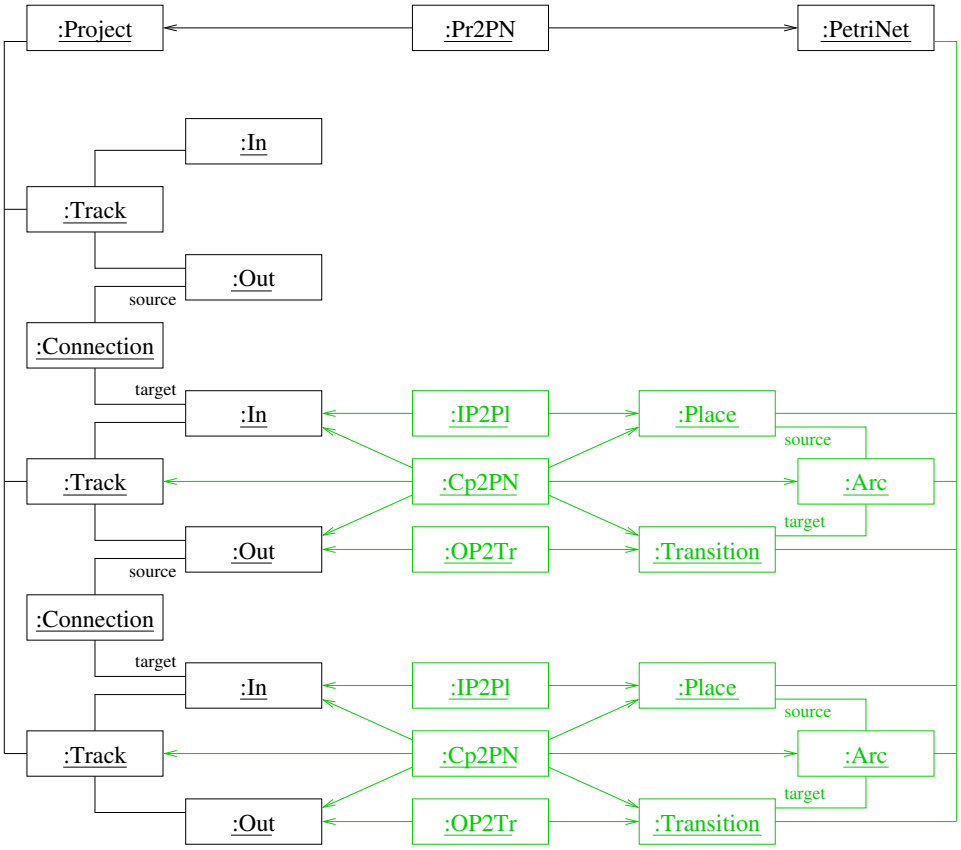

Figure 24: Forward transformation after applying rule for tracks twice

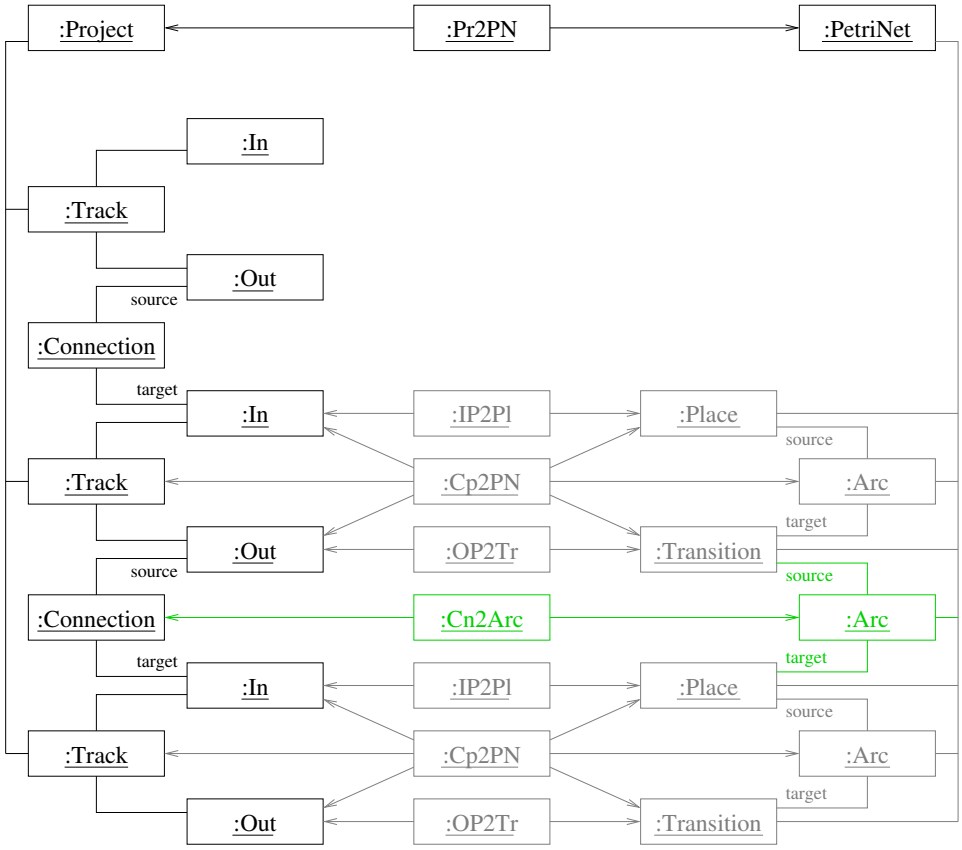

Figure 25: Forward transformation after applying the rule for connections

not have a corresponding project; so we will not be able to fully match the Petri net side of the TGG-rules with the Petri net, and therefore, we will not obtain a complete project model for the Petri net. Secondly, the same Petri net can have different corresponding projects; so the transformation is not deterministic. Thirdly, when trying to match the Petri net side of the TGG-rules with the Petri net model, there might be different choices; for some choices, we might get stuck later during the matching process, whereas other choices might succeed in fully matching the Petri net resulting in a successful transformation. Therefore, the matching process might need backtracking, making the transformation process very inefficient.

For defining the relations between different model types, non-determinism is no problem at all. For some applications, this might be what we want. For making the transformations efficient, however, we typically want to exclude non-determinism. Here, we will not go into the details of these problems. Some of the problems can be solved by a clever matching strategy. Others cannot be avoided. And we need further research in a parsing theory for TGGs which characterizes necessary and sufficient conditions, when the transformation process of a TGG works deterministically and efficiently. This is similar to the parsing theory for classical string grammars such as LL(k)-, LR(k)- or LALR(k)-grammars.

2.4.2 Model Integration

A second application scenario is that we have two models, and we would like to see the correspondences between them. Technically, the setting is very similar to the transformation approach. Since we have models for both domains, we do not need to generate the models – both models are already there. We need to introduce the correspondence node between the root elements of these models as shown in Fig. 26. From there, we match both domains of the TGG-rules on the existing models and introduce the correspondence nodes of the matching TGG-rule. When both models are fully matched, we have established the legal correspondences between the two initial models. We call this *model integration*. In this example, we will end up in the situation shown in Fig. 21 already.

Again, it could happen that we cannot fully match both models. In that case, the models do not fully correspond to each other.

2.4.3 Model Synchronization

The last scenario is similar to model integration, but more general. We assume that there is a model for each domain and that we have correspondences between these models already. But, these models need not be in complete or correct correspondence anymore. Typically, this could happen, when we transform one model into the other as discussed above, and then start modifying one model or even both models independently of each other.

In this setting, the task is to modify both models and the correspondences in such a way that we obtain a complete and correct correspondence according to the rules of the TGG again. We call this scenario *model synchronization*.

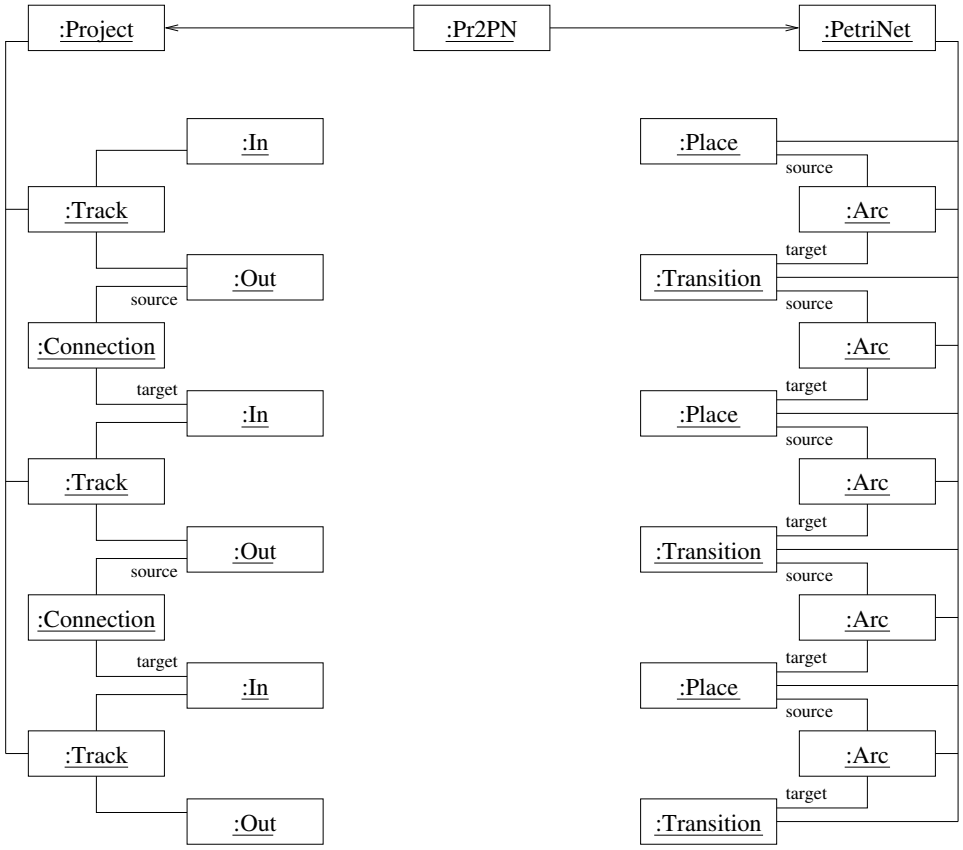

Figure 26: Model integration

Actually, the synchronization task is a bit more involved, because we are interested in a 'best possible matching pair' and with as 'few as possible changes' on both sides. In order to do this, some rules that have been applied earlier need to be taken back, and other rules need to be applied again. In this case, the match of a rule can be partial on both sides and the missing elements in both domains will be added. The exact strategy depends on the application domain and on the last changes made on the two models. Actually, it is the precise definition of 'best possible matching pair' and 'as few changes as possible' that define the strategy.

Technically, model transformation and model integration can be considered as a special case of model synchronization. For a transformation, we start with the situation in Fig. 23 and changes of the source domain are not allowed, but we may introduce elements to the target domain and to the correspondence domain. For the integration of two models, we start with the situation in Fig. 26 and only changes in the correspondence domain are allowed. Since the setting for transformation and integration are much simpler than general synchronization, it is easier to implement them directly. Therefore, we consider transformation and integration as separate scenarios.

2.5 Discussion

In the previous sections, we have discussed the basic idea of TGGs. A set of TGG-rules along with a TGG-axiom define a relation between two kinds of models. What is more, these rules can be made operational in different application scenarios: model transformation, model integration, and model synchronization.

The definition of the relation between two models is driven by the structure of the model. In our example, we have one rule for each concept of the project, as informally shown in Fig. 9. Formally, this relation is defined by the TGG-rules in Fig. 11–14 in the abstract syntax of the two models. Though a bit more verbose[7], the TGG-rules are a way for defining this relation in a *local* and *declarative* way.

This local definition of the relation between different models has several nice implications. Firstly, the TGG-rules can be made operational in the above application scenarios. Secondly, the transformation works in both directions, forth and back; i. e., the definition is *bi-directional*; actually, TGG-rules also work in the model integration and model synchronization scenario. Thirdly, the approach works *incrementally*: for example, let us assume that we have a pair of corresponding models as shown in Fig. 21 already; next, someone adds a train to the last track component as shown in Fig. 27. Then, the TGG-rule for trains can be applied locally at this point and will add the missing parts to the Petri net and the correspondence graph (as shown in green in Fig. 27). Fourthly, the local definition of the relation between the two models strongly resembles the style of *structural operational semantics* (SOS), which helps to

[7]Currently, we are working on a tool which will allow us to specify the rules in graphical syntax as shown in Fig. 9, which will be discussed in more detail in Sect. 4.2.

verify the semantical correctness of the relations between the models.[8]

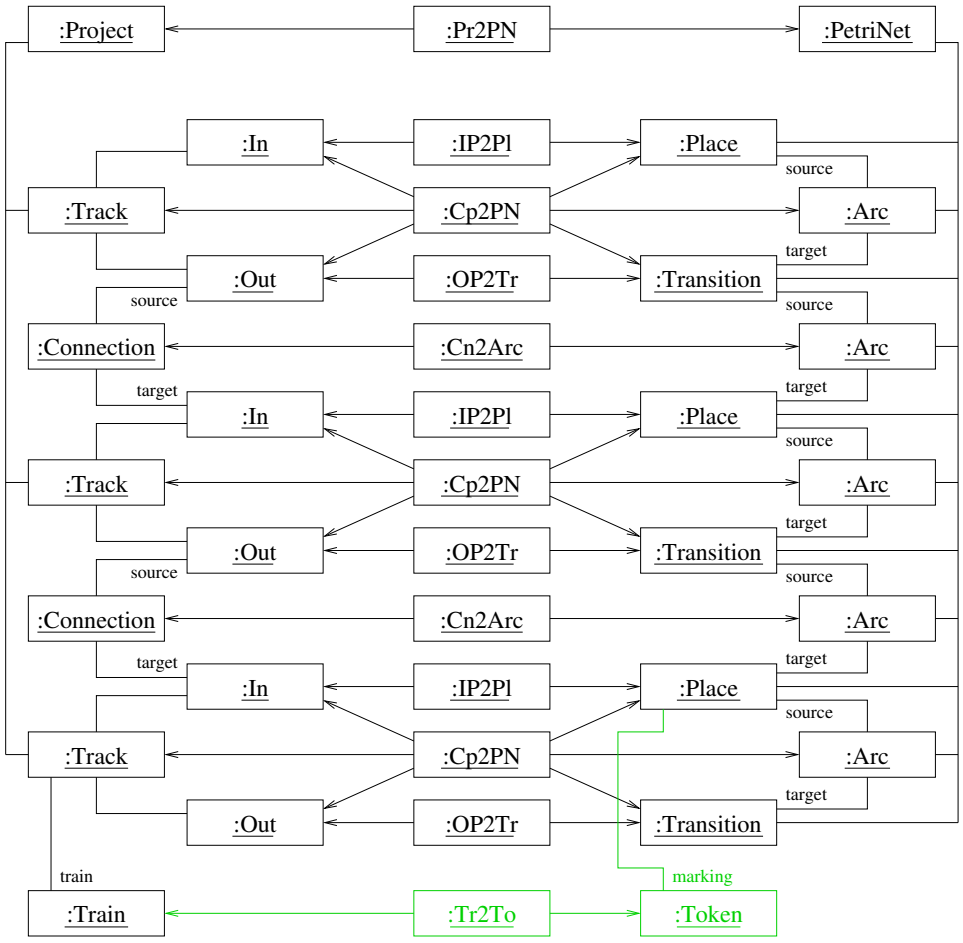

Figure 27: A local modification and its incremental transformation

In order to make TGGs work in practice, however, we need some extensions. In the literature, there are many proposals for extensions, modifications, and variations of TGGs. For example, we need to propagate the values of attributes between the models, we need to consider some additional context not belonging to the meta-model of either model, we would like to have negative conditions, etc. Unfortunately, we need to be very careful in order not to spoil the nice properties and the spirit of TGGs by introducing these concepts improvidently. Therefore, we will deal with these concepts in a separate section. Here, we gave a clear exhibition of the idea and the spirit of TGGs – unspoilt by any additions.

[8] In this paper, we do not cover verification. For more ideas on the verification of TGG-transformations, we refer to [16, 31].

3 Advanced Concepts

In Sect. 2, we have discussed the basic idea and the underlying principles of TGGs. We did not discuss some advanced concepts of TGGs, which are necessary for using TGGs in practice. For example, we need to transform *attributes* of nodes resp. objects, and, for some transformation examples, we need TGG-rules with some additional *constraints* such as the non-existence of some objects, sometimes called *negative nodes*.

The reason for not introducing these advanced concepts along with the basic concepts is the following: Attributes and constraints are important concepts, and are present in many implementations of TGGs. The problem, however, is that, often, they are introduced in a way that breaks the basic principles of TGGs. One of the contributions of this paper is to introduce these concepts in such a way that they do not break the basic principles of TGGs. In order to motivate the way we deal with attributes and constraints, we focused on the principles first. Now, we will discuss the additional concepts in the light of these principles, we will show the problems with the ad-hoc definitions of these concepts, and then present a clear way of defining these concepts.

3.1 Attributes

In UML, object diagrams may have attributes, and techniques for model transformation, integration, and synchronization must take care of attributes too. For example, the components of our project have an attribute name, and there are attributes for the names of the nodes of a Petri net too. Up to now, we did not transform these attributes. But, we might wish to transform these names accordingly. For example, if the name of a track component is $c1a$, we might wish to name the corresponding place of the Petri net $p1a$ and the corresponding transition $t1a$.

The straightforward way of dealing with attributes is to introduce assignments within the TGG-rules. Figure 28 shows such an extension of the TGG-rule from Fig. 11 with the corresponding assignments. The idea is that, when transforming a component project to a Petri net, the place and the transition are named according to the name of the component. In order to refer to the name of the component object, the component now has a unique identifier in the TGG-rule: *cp*. This way, we can refer to the name attribute of the component by *cp.name*. Using Java notation, the method call *cp.name.substring(1)* strips the leading character "c". The assignment `name = "p"+cp.name.substring(1)` takes care of assigning the corresponding name to the place – with a preceding "p". Analogously, the assignment for the transition takes care of assigning the corresponding name to the transition. In order to make the transformation work in the reverse direction, there is an assignment to the name attribute at the component object. It uses the name of the place attribute, strips the first character and adds the prefix "c".

Most implementations of TGGs use this kind of assignments for taking care of the attribute values. And this approach works fine as long as we are interested in forward or backward transformations only. For integration, these

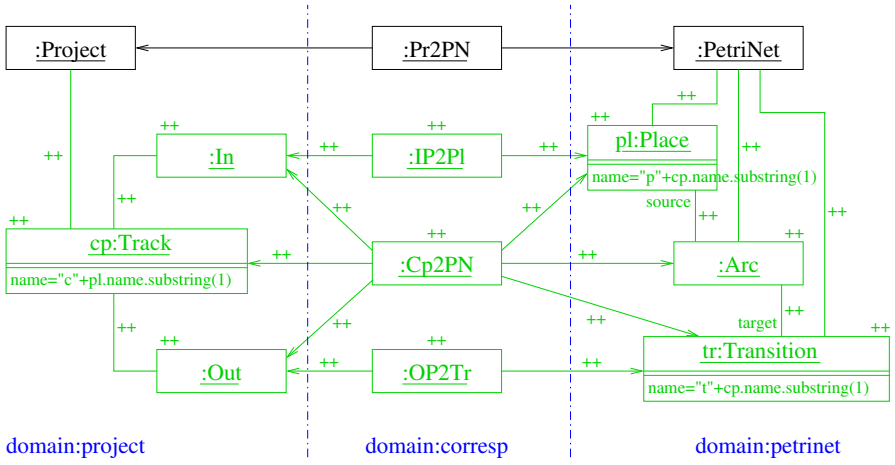

Figure 28: Ad-hoc way for dealing with attributes

assignments do not work any more, rather there should be comparisons. And some implementations add another operation for the integration scenario or just interpret the assignment as a comparison when applied in the integration scenario. Even worse, the assignments do not work when we generate pairs of models by applying TGG-rules starting from an axiom, which is how we defined the semantics of TGGs. This shows that the straightforward way of dealing with attributes breaks the principles of TGGs. Moreover, we can easily have TGG-rules with assignments in such a way that the transformations in forward-direction and in backward-directions are inconsistent – they do different things in different application scenarios. The reason is that we have two (or in our example) even three assignments for expressing a single idea: the component, the place, and the transition should have, basically, the same name – only prefixed by a "c", a "p" or a "t", respectively.

Therefore, we introduce a concept for dealing with attributes in a clearer and more uniform way. The main idea is that the actual name or rather the suffix (stripping the first character) is an attribute of the correspondence node of class *Cp2PN*. Actually, we introduced that attribute already in the meta-model for the correspondence objects in Fig. 17 – we called it *suffix*. The values from either domain are now derived from this suffix attribute. The relations of the name attributes of the different nodes are then defined by *constraints*, which relate the attributes of the objects of one domain and the correspondence domain. This is shown in Fig. 29. The rounded boxes represent these constrains, where each constraint can be read as an assignment, which constructs the corresponding name from the suffix attribute of the correspondence node.

Actually, there is no restriction on the exact constraints imposed on the different attributes from a semantical point of view. It is only required that, when we have a matching pair of models, all constraints are met. For some application scenarios, however, some restrictions need to be imposed on the constraints in order to efficiently execute transformations and synchronizations. For example, the expression on the right-hand side of the "assignment" should be invertible. This will be discussed with the tool support.

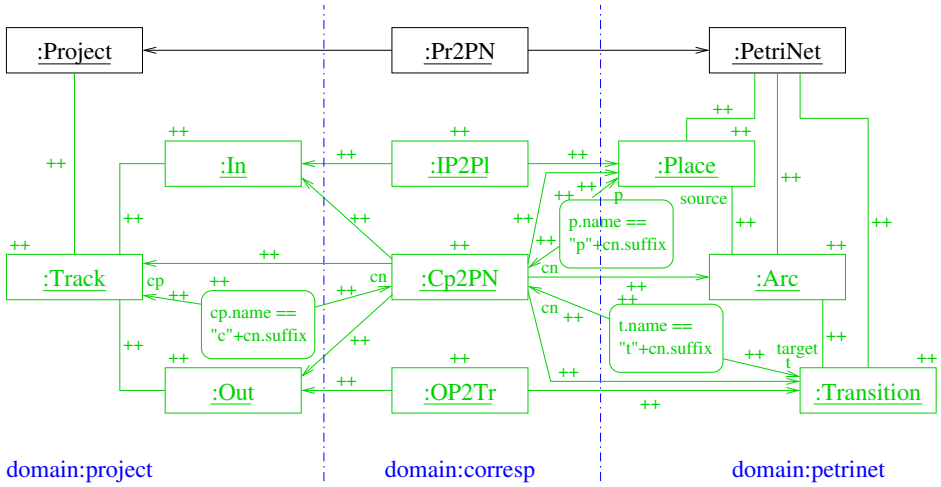

Figure 29: TGG-rule with attributes

3.2 Constraints

In the previous section, we have used constraints already for expressing the relations between values of attributes. In this section, we generalize the concept of *constraints*. Again, we start with a motivation and a discussion of some issues and problems related to constraints. In the end, we will present the way, we deal with constraints.

Let us consider an example first. In Fig. 15, we introduced a rule that transforms a train associated with a component into a token in the corresponding place of the Petri net. Now, for the sake of an argument, let us assume that we want to transform a train associated with a component to a token on a place only if the component has no incoming connections (i. e., the initial components) resp. if the corresponding place has no ingoing arc. Figure 30 shows the extended rule, where the constraint imposes the additional condition on the place by an OCL expression: the set of *in* arcs should be empty.

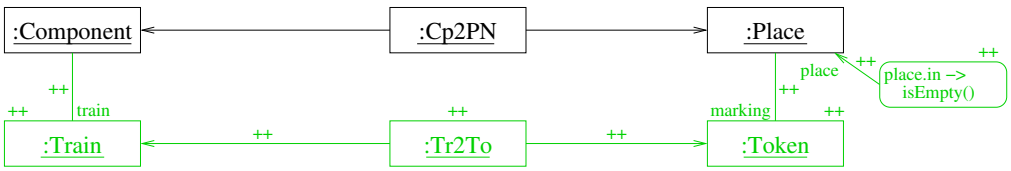

Figure 30: Extended TGG-rule for a *Train*

The meaning of this constraint seems to be obvious. But, actually there are different interpretations, depending on the application scenario. The differences come from the time at which the constraint is checked. For example, in the transformation scenario from a component project to a Petri net, it could be that we first generate the Petri nets for the two track components as shown in Fig. 24. In this situation, both places do no not have incoming arcs, and therefore, the constraint is true at that time. If we had a train on these components,

we might figure that the TGG-rule for the train could be applied. But later on, the TGG-rule for a connection is applied, which adds an incoming arc to one place (cf. Fig. 25) and finally both of the above places will have an incoming arc (cf. Fig. 21). So, in the end the constraints are violated. This shows, that it makes a difference at which time the constraints are evaluated. In particular, this makes a difference between the forward and backward transformation, the integration and synchronization, and between the actual generation semantics of TGGs.

Of course, the semantics and the relation between models should be the same for all application scenarios. Therefore, we introduce a uniform concept: The idea is that all constraints are added while applying the rules and while constructing the models. But, the constraints will be evaluated only after the full model is constructed. The relation between the two models is only valid, if all constraints evaluate to true in the final result. Of course, this is only the definition of the semantics of TGGs; in practice, depending on the application scenario, the constraints can and will be considered to chose the TGG-rule to be applied and to resolve between different choices of applicable TGG-rules. This, however, is an implementation issue and not a semantical issue.

Due to this a-posteriori semantics of constraints, all application scenarios will produce valid relations only. Moreover, the actual constraint in a *constraint* node can be any OCL expression that refers to the objects it is attached to. Restrictions are only necessary in order to make implementations of the transformations or synchronizations more efficient.

3.3 Reusable Nodes

Up to now, TGGs require that all parts of the two models are completely captured by the TGG-rules, and all parts are completely generated. Sometimes however, there are parts of models that should neither be generated nor be changed by the transformation or synchronization of two models. Still, the model is related to these parts and the relation between the two models must take these parts into acount. In order to properly deal with this problem, we introduce two additional concepts: *reusable nodes* and *modes*. We will discuss reusable nodes in this section; modes will be discussed in the next section.

In order to illustrate these concepts, we extend our example from Sect. 2.3. In the meta-model for projects (see Fig. 8), we had a separate class for each type of component: *Train* and *Switch*. Actually, Component Tools is much more flexible [28]: the possible types of the components are not fixed, but can be defined in a so-called *component library*. The type of a component is not encoded in the class, but by a reference to the *component type*, which is defined in the component library. Figure 31 shows a simplified version of the corresponding meta-model. A component library consist of some component types. Each project is associated with exactly one component library and each component is associated with exactly one component type. In a transformation, this library should not need be transformed, but the references to it must be taken into account.

The basic idea for the rules for transforming these kinds of projects is the

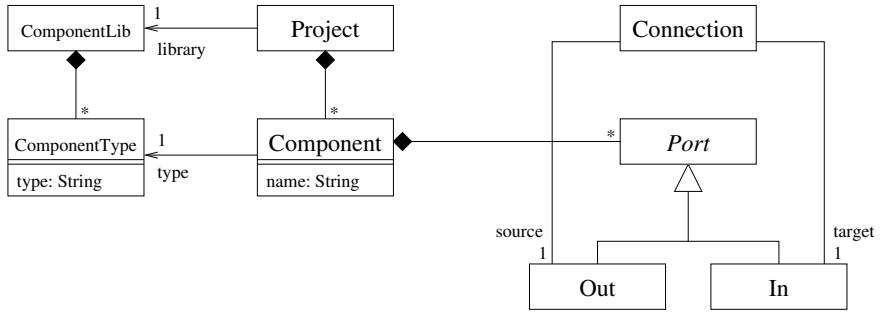

Figure 31: The extended meta-model for projects and libraries

same as before. The only difference is that the type of a component is now encoded in the reference to a type in the library. The TGG-rule for transforming a track to a Petri net (as originally defined in Fig. 11) is now modified as shown in Fig. 32. The component now needs to refer to a component type from the component library, and the constraint says that the type is "Track".

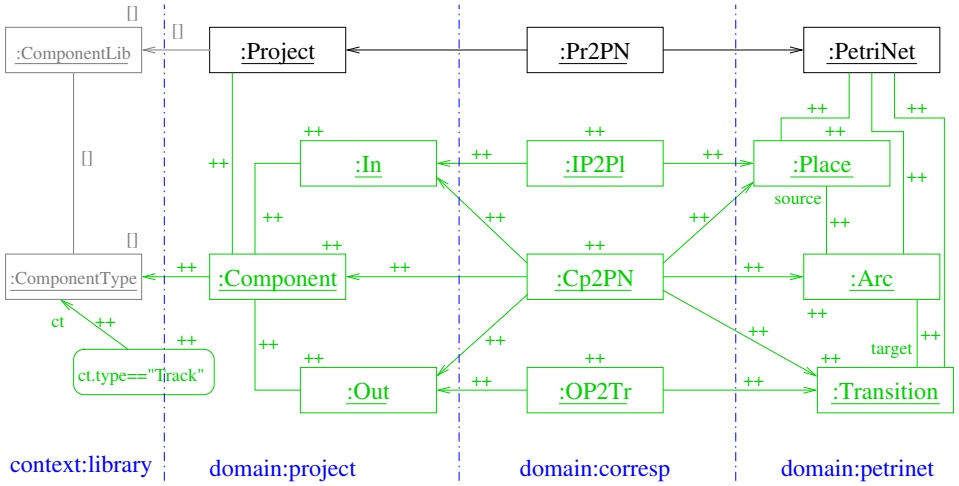

Figure 32: The new TGG-rule for component *Track*

During a transformation or synchronization, the component library should not be changed. But, the library, the type, and a reference need to be there. And it needs to be checked that they are there. Note that the component types do not need to be generated, but can be reused over and over again form the already existing nodes with the required properties. Therefore, we call them *reusable nodes*. Graphically, these nodes are represented in gray and with a label []. The semantics of the reusable nodes is that they can be newly generated or reused in an arbitrary way. The gray color reflects the fact that, semantically, each reusable nodes could be either black (a node from the left-hand-side of the TGG-rule, i. e., it is reused) or green (a node from the right-hand-side of the TGG-rule, i. e., it is newly generated), which can be chosen whenever the rule is applied. This interpretation shows that actually, reusable nodes are not strictly necessary. We could replace it by exponentially many TGG-rules

with all possible 'colorings' of the gray nodes. But, the concept of reusable nodes helps reducing the number and the complexity of TGG-rules. In more complex examples, the rules without reusable nodes might become really messy. Reusable nodes allow us to have simpler TGG-rules and to concentrate on the essence of the relevant models.

3.4 Modes

As discussed above, the reusable nodes can be newly generated or existing nodes with the same property can be reused. In the above example, however, the transformation should not create new nodes in the component library. This is where the concept of *modes* comes in. The domain *library* can be used and accessed in the transformation, but it should not be changed. So this domain is used in mode *access*.

The mode of the other domains depends on which of the scenarios we use: For a transformation from a Component Tools project to a Petri net, the domain *project* has mode *read* and the domains *corresp* and *petrinet* have mode *write*. For the synchronization scenario, all domains *project*, *corresp*, and *petrinet* have mode *write*. For the integration scenario, the domains *project* and *petrinet* have mode *read* and the domain *corresp* has mode *write*.

Actually, the application scenario can now be precisely defined by assigning a mode to each domain of the TGG. Mode *access* says that this domain may be accessed, but not modified. Mode *read* says that this domain also may not be modified; by contrast to mode *access*, mode read requires that at the end of the transformation every element of that domain was actually read, so that the domain was covered completely. In mode *write*, the domain may also be changed and, upon termination, all elements of the domain have been used.

This way, the application scenarios as defined in Sect. 2.4 can be defined in terms of modes for each domain. Note that there could be even more refined modes, which allow only adding or only removing nodes or only changing attributes for more refined definitions of application scenarios. But, this needs further investigation and is beyond the scope of this paper.

3.5 Short hand Notations

In the previous sections, we have introduced some advanced concepts for TGGs. In fact, some of these concepts have been first introduced in a different way and in different notations. Since some of these notations are well-known in the TGG community, we introduce them as a short hand notation here. But, the semantics of these short hand notations will be defined by translating them to the concepts defined in the previous sections.

3.5.1 Assignments

As discussed in Sect. 3.1, the values of attributes of newly created nodes is often defined by an assignment for this attribute, which belongs to that node (see Fig. 28). Later we introduced a more general concept of *constraints*.

Every *assignment* to some attribute of some node can be easily replaced by a constraint. For example, the assignment `name = "p"+cp.name.substring(1)` in node *pl* of the TGG-rule in Fig. 28 could be replaced by a constraint node attached to node *pl* and node *cp* with the following constraint:

```
pl.name == "p"+cp.name.substring(1)
```

And all the other assignments in this TGG-rule could be replaced by an analogous constraint.

Note that in Fig. 29, we did not use this straight-forward translation of these assignments. Rather we distilled the common parts of the different *name* attributes in an extra attribute *name* of a correspondence node and derived all the other names from there. Moreover, this makes the implementation of a transformation or a synchronization engine more efficient and helps avoiding inconsistent constraint, which could arise from a pair or a set of inconsistent assignments.

3.5.2 Negative Links

In some situations, we would like to ensure that, in a pair of corresponding models, some links are not there. To this end, there are different notations using crossed out links or nodes. Since there are some ambiguities when crossing out nodes, we introduce a short hand for crossing out links only. All other situations can be expressed with constraint nodes, if necessary.

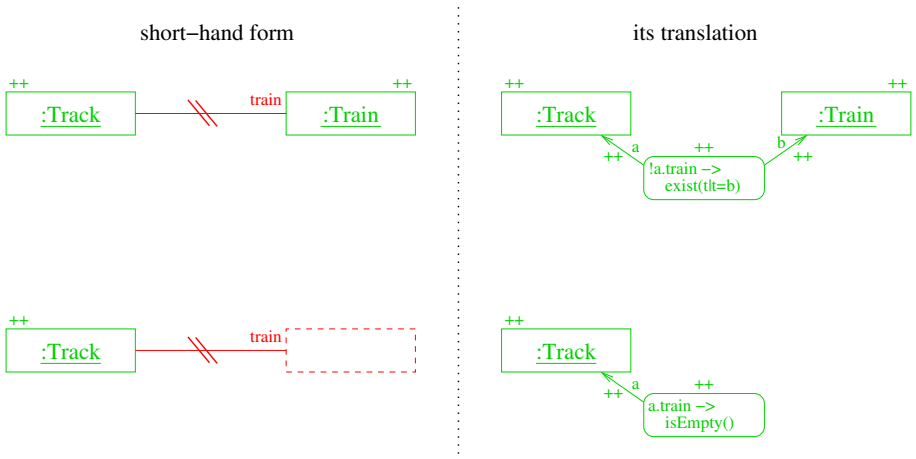

Figure 33: Two parts of a TGG-rule with negations and the corresponding constraints

Figure 33 shows two parts of a TGG-rule with negations in them on the left-hand side. The right-hand side shows their meaning in terms of a constraint. The example on the top shows a crossed out link between two nodes–which is also shown in red. The meaning is that no such link should exist. This is expressed exactly by the constraint shown on the right. The example on the bottom says that there should be no link to any other node. This non-existing node is shown in red as a dashed box. Again, this can be expressed by the constraint shown on the right-hand side.

4 Usability

As we have seen in the previous sections, TGGs can be used to define the relation between two (or more) types of models and to transform such models into each other and to synchronize such models. In practice, there will be some 15 to 20 rules and each rule has some 10 to 40 nodes. TGG-rules with some 40 nodes are quite hard to edit, check, debug, and revise manually. Therefore, we need methods and tools for making the editing and design process for TGG-rules more comfortable.

In this section, we discuss two approaches to design the TGG-rules in a more comfortable way: The first approach automatically generates the TGG-rules from some pairs of corresponding models. The second approach simplifies the editing of the TGG-rules by not using object diagrams with the abstract syntax, but the graphical syntax of the models directly. These approaches are discussed below.

4.1 Specification by Example

Up to now, a correspondence mapping between two or more models has been specified using the abstract syntax defined by the meta-models of the involved modeling languages. However, meta-models are not always as simple and easy to understand as in our running example. Rather, in most cases the meta-models are quite large and contain many different concepts which often lack an obvious link to their concrete representation in the modeling language (cf. the meta-model of the *Unified Modeling Language* (UML) [36]). As a consequence, a meta-model based definition of a correspondence mapping between two or more models becomes quite complicated.

In order to ease the specification of a correspondence mapping between models, we propose to specify the correspondence mapping by the use of example pairs given in the concrete syntax of the involved modeling languages. From the given example pairs, we synthesize the TGG-rules automatically.

In the following, we present the basic idea of our approach and the requirements for its implementation which was worked out in full detail in a masters thesis [15]. Then, we informally describe the basic rule synthesis algorithm, its extensions concerning the given sequence of example pairs as well as necessary extensions in order to handle the introduced concepts of attributes, constraints, and reusable nodes. We close this subsection with a discussion of our specification by example approach and some recommendations for its application.

4.1.1 Basic Idea

The basic idea of our approach is that the user specifies the relation between two model types by providing a set of example pairs. An example pair consists of two sample models which correspond to each other – each from the domain of the involved modeling language. The correlation of the sample models in the example pair defines a semantic relationship resp. correspondence between both models. In the upper part of Fig. 34, an example pair defining the correlation

between a project and a Petri net is given. The example pair is defined using the concrete syntax of the involved modeling languages. For the definition of a correspondence mapping the user's knowledge about the modeling language is sufficient.

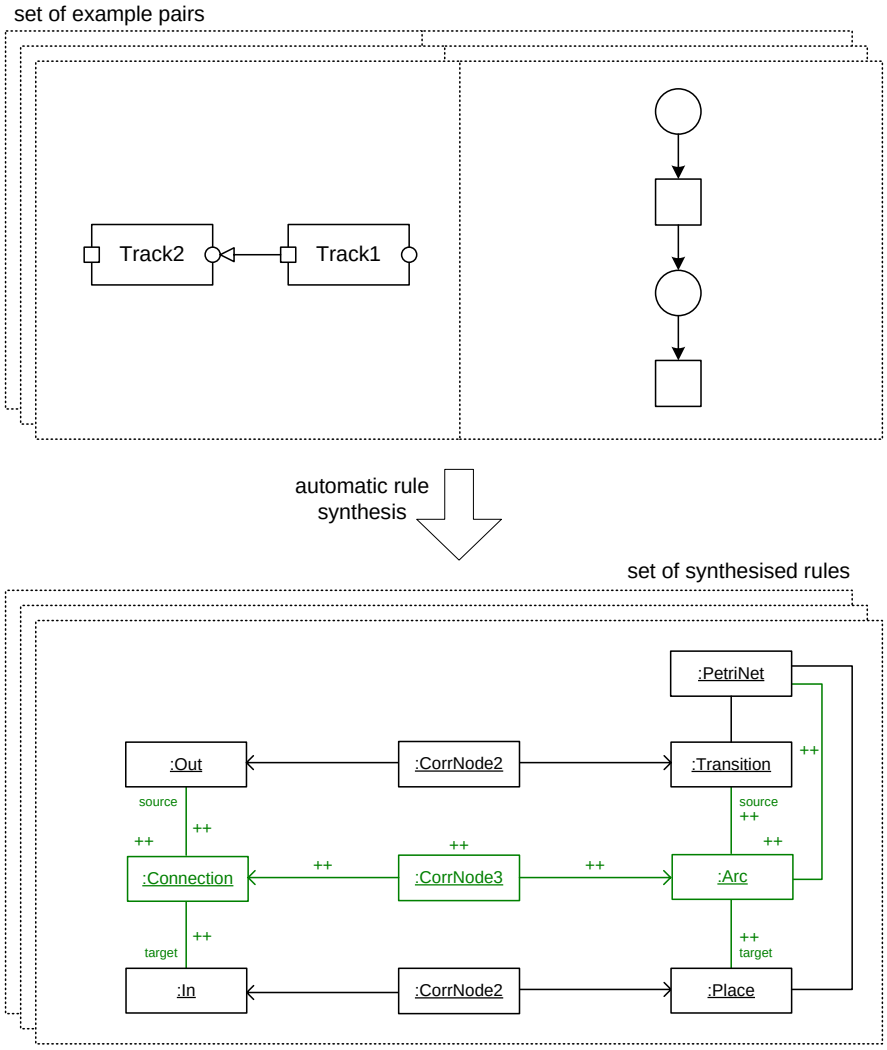

Figure 34: Overview: Specification by Example

In order to utilize the given set of example pairs for the presented application scenarios like model transformation, integration, or synchronization, we synthesize the necessary TGG-rules from the set of example pairs automatically. A necessary prerequisite for the automatic rule synthesis is the translation of the example pairs into the TGG-formalism. In Fig. 35, the initial translation of an example pair into the TGG-formalism is shown. The example pair relates a track to its Petri net representation which consists of a place, a transition and an arc from the place to the transition. In the top of Fig. 35, the example pair in its graphical and abstract syntax representation is shown; in the bottom the same example pair is shown after the translation to the TGG-formalism.

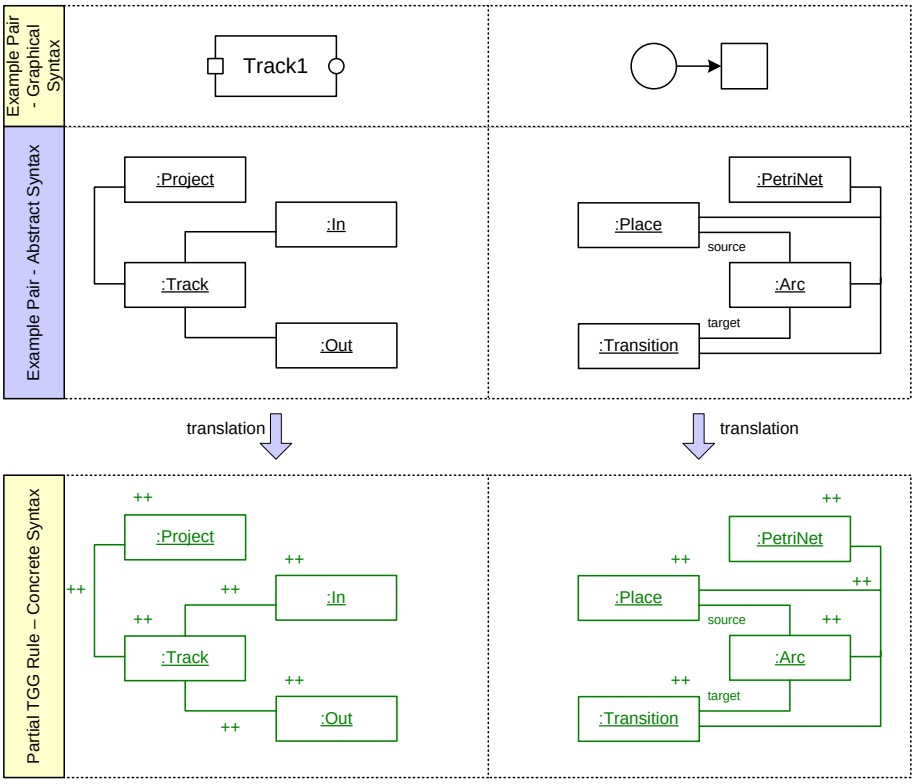

Figure 35: Initial translation to the TGG-formalism

As shown in Fig. 35, the graphical representation of a TGG-rule is quite similar to the abstract syntax representation of the example pair. However, there is a fundamental difference between the abstract syntax representation of the example pair and the graphical representation of a TGG-rule. For example, in a TGG-rule the nodes and links can have additional ++ labels. Or, we can mark the nodes as reusable and assign some constraints to them.

4.1.2 Basic Rule Synthesis Algorithm

In order to explain the basic rule synthesis algorithm, we exemplify our synthesis algorithm by the help of our running example. We start with an empty set of example pairs and extend the provided set of example pairs step by step.

First Example Pair The simplest example pair consists of an empty project which is mapped to an empty Petri net. It is provided by relating an empty project diagram to an empty Petri net diagram. In the first step, this example pair is translated to the TGG-formalism as shown in the previous subsection. According to the meta-models shown in Fig. 2 and Fig. 8, an empty project is represented by an object of type *Project* and an empty Petri net respectively as an object of type *PetriNet*. During the translation, both objects are assigned to their domains. In addition, the objects are marked with ++ labels, i.e., initially they are marked to appear only on the right-hand side of the grammar rule.

After the translation into the TGG-formalism, the synthesis checks if some already synthesized rules exist which have to be checked against this new example pair. Since this is the first example pair, no further rules have to examined. Therefore, our synthesis algorithm introduces a correspondence node between the *Project* and *PetriNet* objects. In Fig. 36, the extracted TGG-rule is shown.

Figure 36: Step 1

At the moment, the extracted TGG-rule has an empty left-hand side and provides only objects on the right-hand side which are therefore marked with ++ labels. Hence, the synthesized rule does not correspond to the so far known structure of TGG-rules. In fact, it is similar to the axiom shown in Fig. 16. For the further processing, the rule will be left as is. However, at the end of our rule synthesis algorithm, the ++ labels will be removed which will result in the previously introduced axiom shown in Fig. 16. We recognize that this is an axiom by the fact that the rule contains only one correspondence node.

Second Example Pair We proceed with the rule synthesis by providing a second example pair. The second example pair relates a track element to a Petri net as shown in the top of Fig. 35. Once again, the synthesis starts with the translation of the example pair to a TGG-rule. The result of this translation is shown in Fig. 37.

Figure 37: Step 2

Now, the synthesis algorithm checks if some other rules exist that could be matched to the extracted object structure of the example pair. In our case, only the rule shown in Fig. 36 has to be considered. The algorithm matches the objects from the already synthesized rule with the new example pair according to their domains. This is similar to the integration scenario presented earlier except for the fact, that the correspondence node is not considered during the matching procedure. Since the correspondence node of the synthesized rule is omitted, a valid match can be found as shown in Fig. 38.

Figure 38: Step 3

Due to the valid matching, the ++ labels are removed from the matched objects, i.e., the objects become black. In addition, the correspondence node of the matched rule is added to the considered object structure of the new example pair. In order to obtain a valid TGG-rule, the ++ label of the added correspondence node is deleted. The intermediate result of the rule synthesis so far is shown in Fig. 39.

The matching procedure is executed for each synthesized rule as long as valid matchings with existing rules can be found. In the example, the axiom matches only once. Therefore, the algorithm proceeds and since this is the only rule synthesized so far, no further rules have to be checked. In the last step, a new correspondence node is added to the object structure under consideration. This correspondence object connects all remaining objects from both domains which are marked with the ++ labels. In order to obtain a valid TGG-rule, the correspondence node and its links are also marked with ++ labels. Altogether we obtained the TGG-rule shown in Fig. 40.

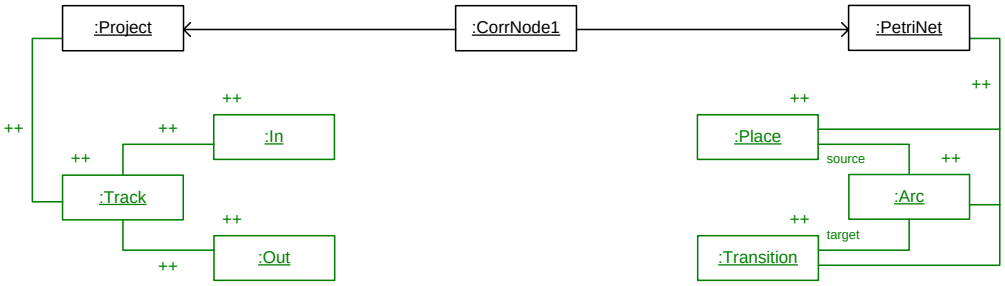

Figure 39: Step 4

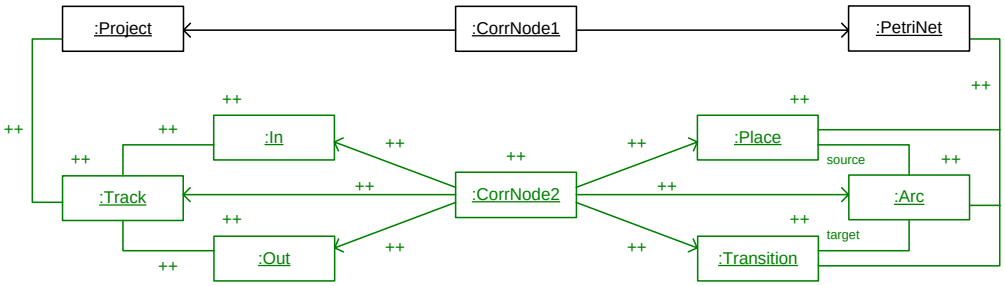

Figure 40: Step 5

The synthesized rule in Fig. 40 resembles the hand-crafted rule in Fig. 11. However, there are two differences between the hand-crafted rule and its automatically synthesized counterpart. The first difference is the type of the correspondence nodes. In manually designed rules, the types of the correspondence nodes can be modeled explicitly at the discretion of the rule designer. For example, in our hand-crafted version of the rule, the correspondence type reflects the established mapping between the involved object types. In contrast to that, in the automatically synthesized rule, the types of the correspondence nodes are generated automatically by numbering them consecutively. This numbers reflect the order in which the rules have been synthesized.

The second difference is the number of correspondence nodes. In the hand-crafted rule, three correspondence nodes are used for expressing the mapping of the track component as well as the ports to the corresponding Petri net elements. In the synthesized rule only one correspondence node is generated. Although this is quite sufficient for our example, the manual design of TGG-rules allows to establish a more fine-grained mapping structure, which increases the understandability of the TGG-rules and sometimes makes the transformations more efficient.

Third Example Pair Up to now, the given example pairs have been quite simple – they relate only a few project elements to a corresponding Petri net. So let us consider the more complex example pair shown in the upper part of Fig. 34. The example pair consists of two connected track elements and their appropriate Petri net representation.

As in the previous synthesis steps, the example pair is prepared for the rule

synthesis by translating the underlying abstract syntax representation to the common TGG-formalism. The translated example pair is shown in Fig. 41.

Figure 41: Step 6

Now, the synthesis algorithm has to check if some of the already synthesized rules can be applied to the new example pair. Due to the fact that our new example pair does not have any black objects yet, the only applicable rule is the rule from Fig. 36. In Fig. 42, the rule matching is shown. Once again, the objects *Project* and *PetriNet* are matched successfully to the example pair. Therefore, the ++ labels are deleted from the matched objects in the new example pair and a correspondence node is inserted. The result of this matching is shown in Fig. 43.

Now, the intermediate TGG-rule has some black objects without the ++ labels. Therefore, the TGG-rule from Fig. 40 becomes applicable and can be matched. In fact, it can be matched twice. The first match is shown in Fig. 44. The second match is presented in Fig. 46.

The matched objects – if not already black – are in turn rendered in black, i.e., the ++ labels are removed. In addition, the correspondence nodes are added to the intermediate TGG-rule. The result of the first matching is shown in Fig. 45. The result of the second matching is presented in Fig. 47.

Due to the performed matching, only a few objects marked with ++ labels are left. In particular, this is the *Connection* object in the project domain and the *Arc* object in the Petri net domain. Since no further rules can be matched to the resulting intermediate TGG-rule, the synthesis algorithm inserts a new correspondence node relating these objects to each other. The inserted correspondence node connects two objects marked with the ++ labels. In order to obtain a TGG-rule which can be used for model transformation, integration, and synchronization purposes, the correspondence node is also marked with the ++ label. The resulting TGG-rule is shown in Fig. 48.

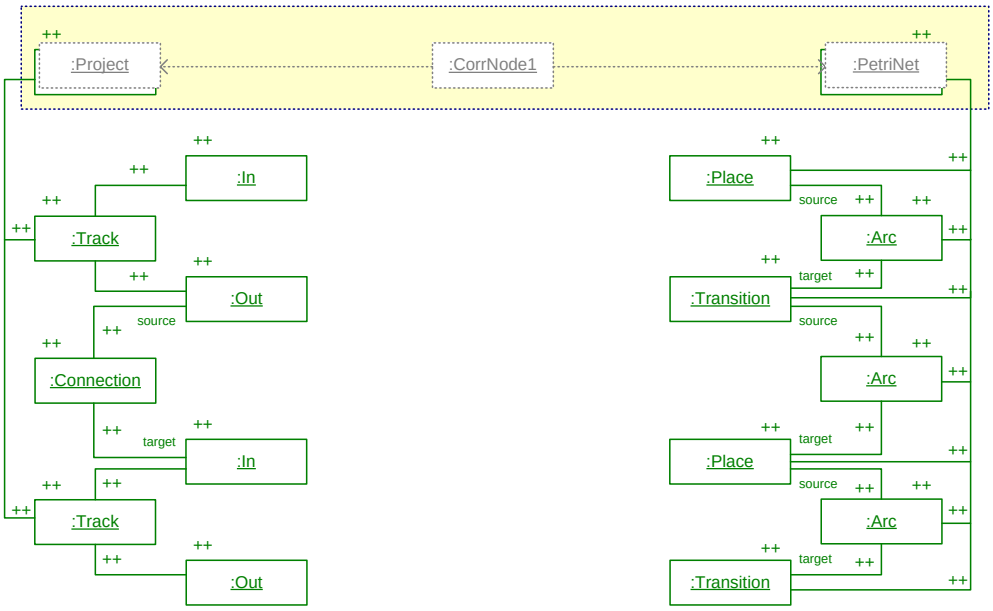

Figure 42: Step 7

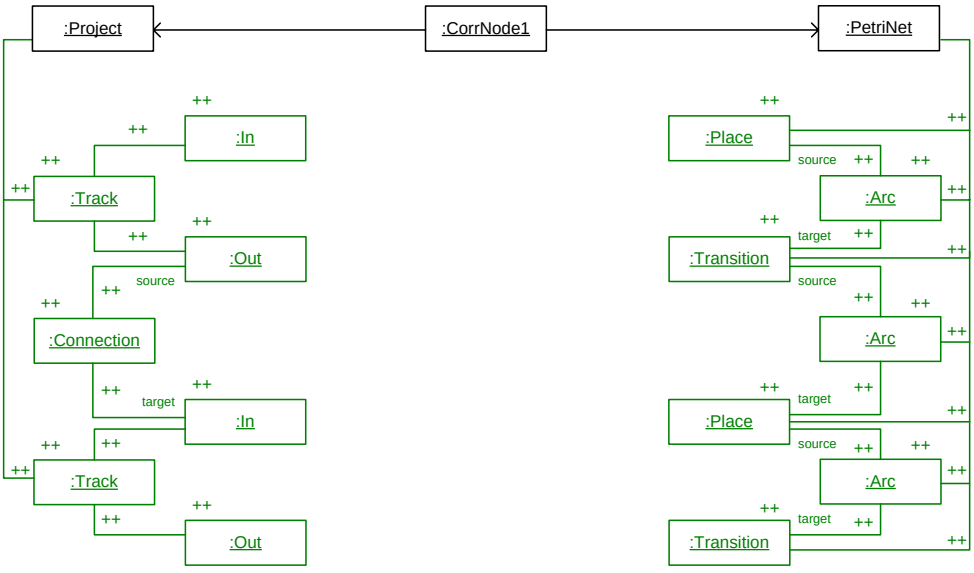

Figure 43: Step 8

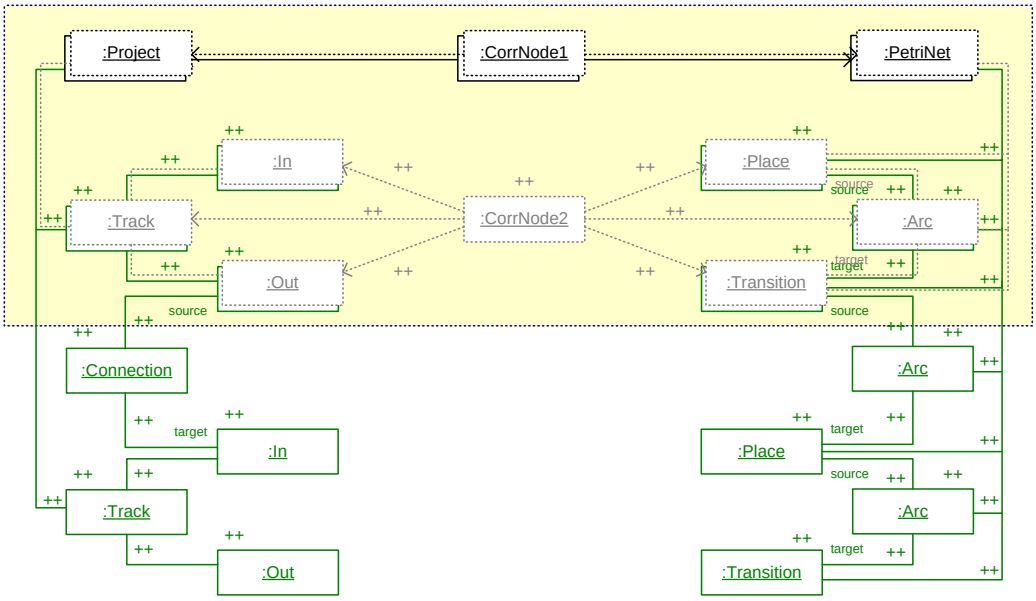

Figure 44: Step 9

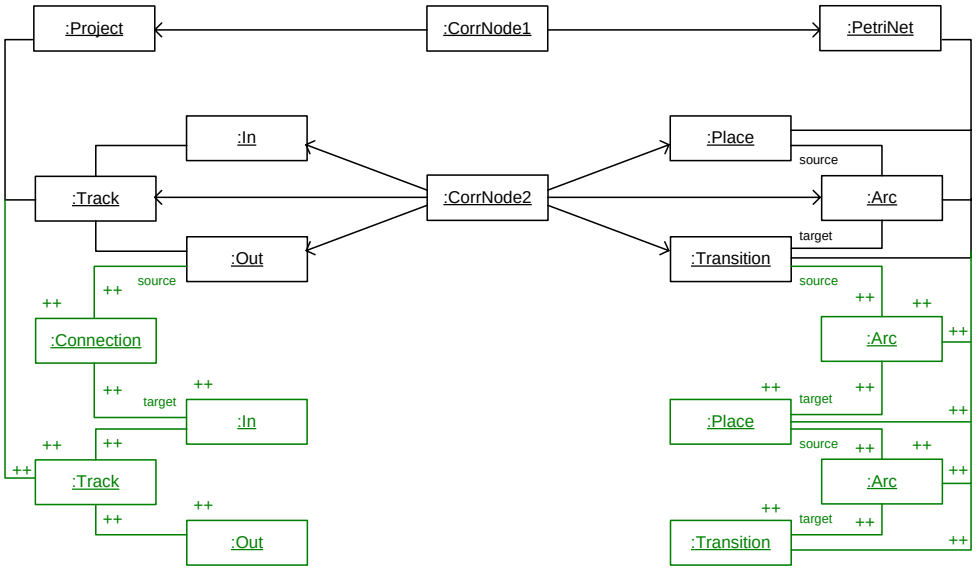

Figure 45: Step 10

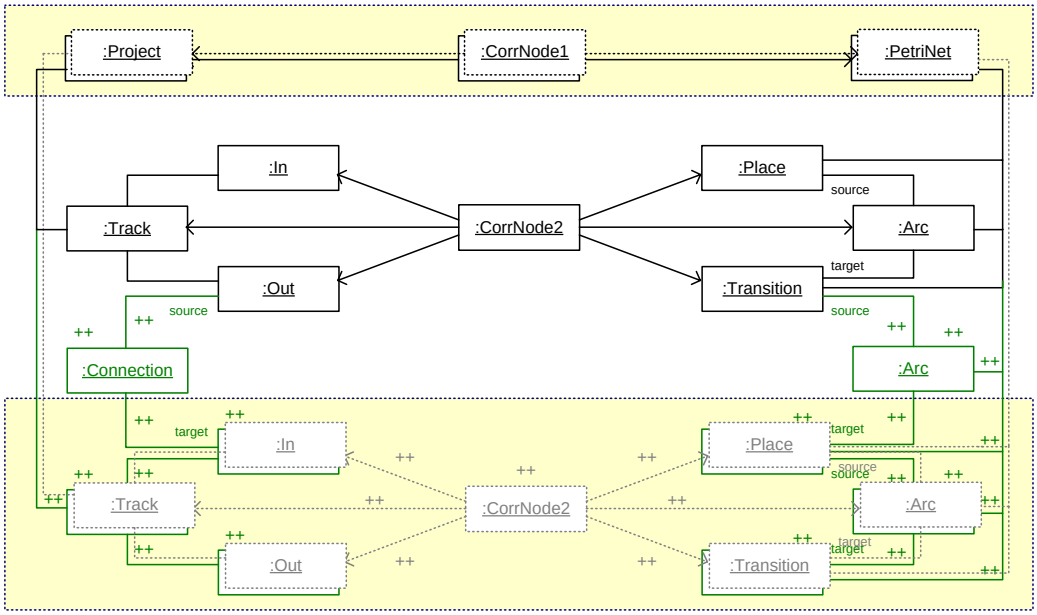

Figure 46: Step 11

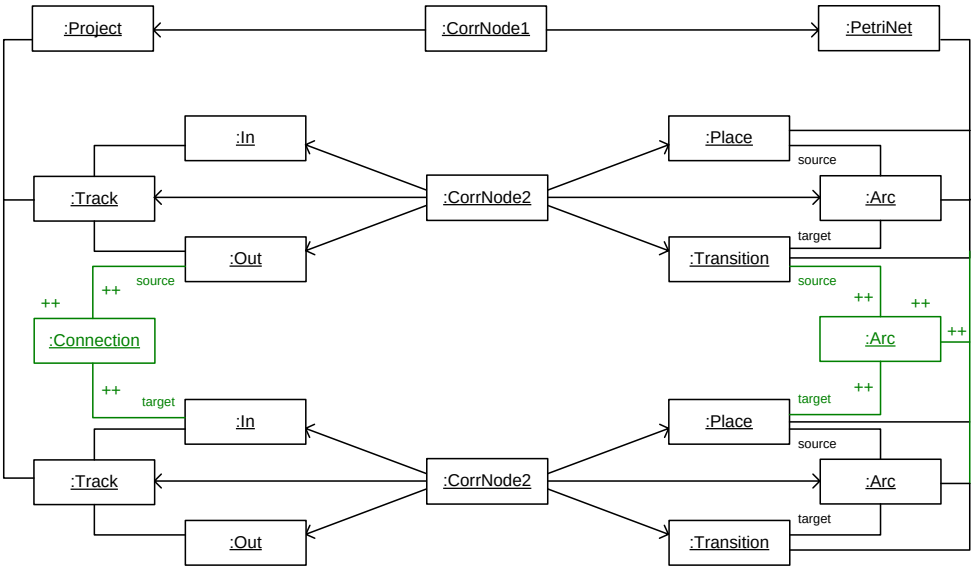

Figure 47: Step 12

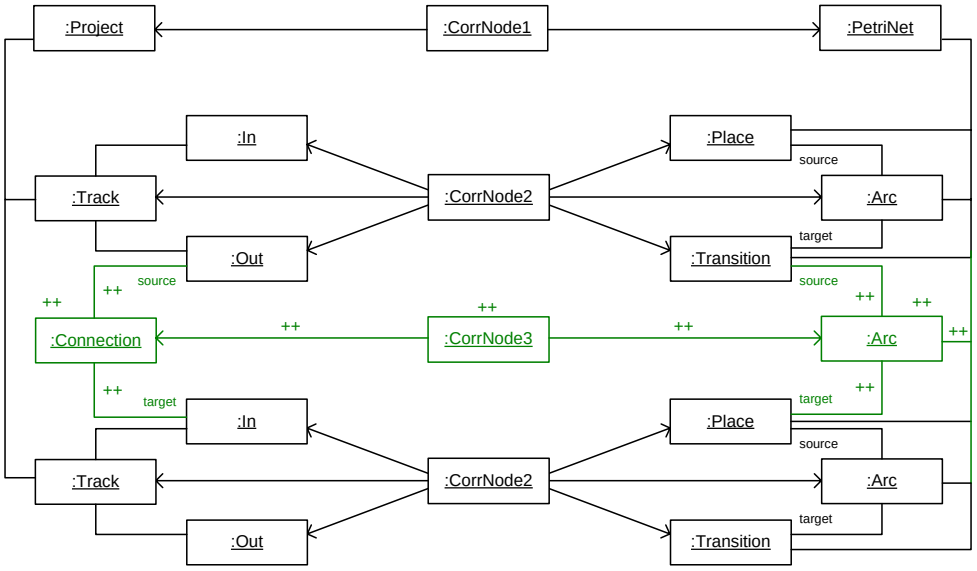

Figure 48: Step 13

The synthesized rule in Fig. 48 conforms already to the well known structure of TGG-rules: it has some black objects and some green objects with ++ labels. However, the synthesized rule contains quite many black objects. During the execution of a model transformation, integration, or synchronization, all the black objects have to be matched to a given model. These additional matching efforts reduce the overall performance and restricts the rule applications. However, due to our experience, not all objects are really needed for the specification of a valid mapping; simply speaking some of them are redundant. Therefore, our algorithm always tries to minimize the number of black object in a TGG-rule using a simple heuristic.

In order to get rid of these redundant objects and to reduce our rule to a minimal set of objects, the synthesis algorithm examines all objects and checks for objects which are not directly connected to green objects, i.e., it searches for objects which have no direct link to an object marked with a ++ label. In Fig. 49, these objects are labeled with a red x. Note that the *PetriNet* object is not marked with a red x since it is directly connected to the *Arc* object.

All marked objects and their incidents links are deleted from the synthesized TGG-rule. In addition, all correspondence nodes without links or with links to only one domain are removed. For example, after the deletion of the *Project* object and its incident links, the correspondence node *CorrNode1* will only have a link to the Petri net domain object *PetriNet*. Therefore, this correspondence node will be deleted as well. The final TGG-rule after the deletion of the redundant objects is shown in Fig. 50.

The automatically synthesized TGG-rule resembles the hand-crafted TGG-rule for a connection presented in Fig. 14. The only difference between the synthesized and the hand-crafted rule is the additional link between the *PetriNet* object and the *Place* object. In fact, in the Petri net model, the *PetriNet* object

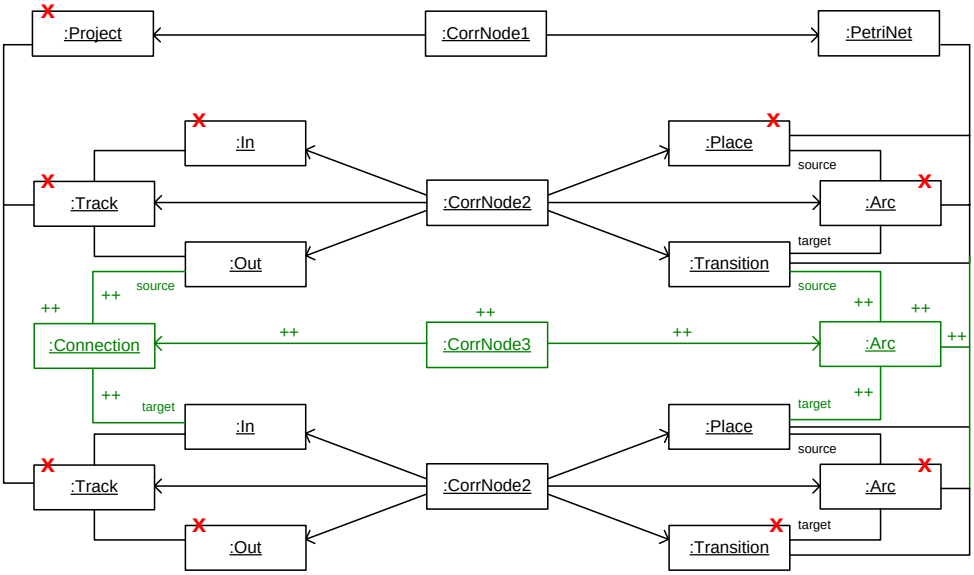

Figure 49: Step 14

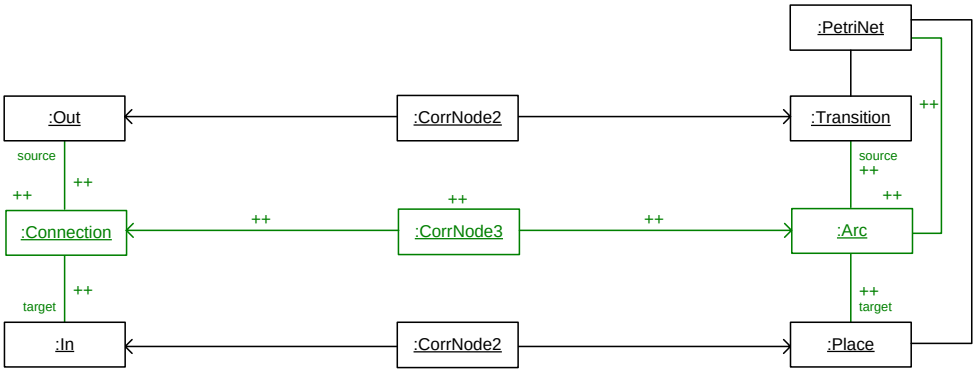

Figure 50: Step 15

has links to *Transition* objects as well as to *Place* objects. Therefore, the synthesized TGG-rule is correct. However, when specifying a TGG-rule manually, the designer is more flexible and can decide to omit some of the links.

In this section, we have exemplified the basics of our rule synthesis algorithm. We have started with an empty set of example pairs and extended it step by step. This iterative synthesis procedure was possible since the example pairs have been provided in a quite advantageous order. In order to be able to synthesize TGG-rules even if the order of example pairs is not such advantageous, or if the set of example pairs is not extended incrementally but given at once, we have to modify our synthesis algorithm in such a way that it is independent from the provided example pair order. This is explained in more detail in the following section.

4.1.3 Example Order

Let us assume that the order of provided example pairs is changed. First, once again the example pair relating an empty project with an empty Petri net is given. Then, the example pair relating two connected tracks and its Petri net representation is provided (cf. upper part of Fig. 34). Lastly, the example pair from Fig. 35 is made available.

From the example pairs given in the new order, our algorithm will once again synthesize three TGG-rules. From the first example pair once again the TGG-rule presented in Fig. 36 is synthesized. From the second example pair the TGG-rule shown in Fig. 51 is synthesized whereas the TGG-rule synthesized from the third example pair is once again the TGG-rule presented in Fig. 39.

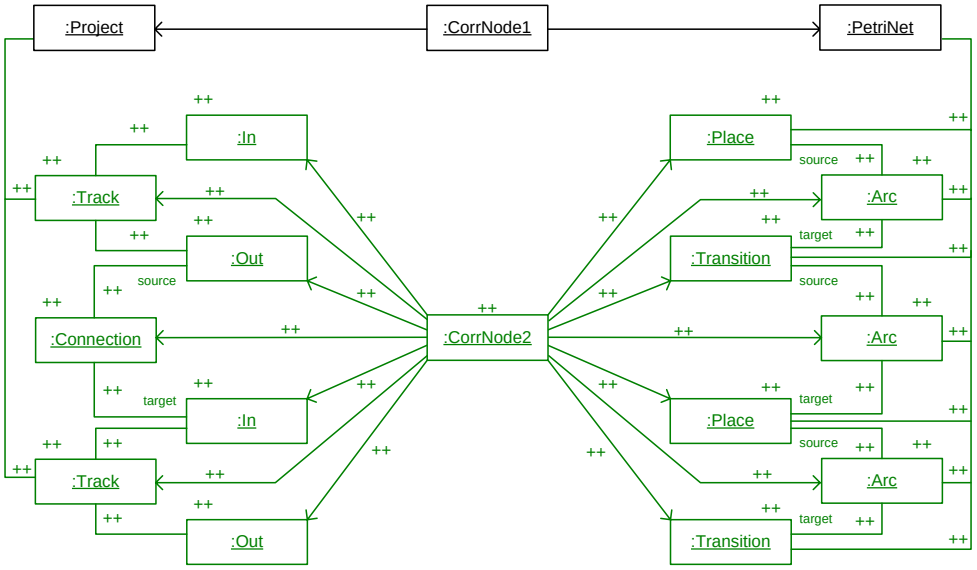

Figure 51: TGG-rule resulting from second example pair

The TGG-rules shown in Fig. 50 and in Fig. 51 have been synthesized from the same example pair. Altogether, the rules are quite different. This results from the changed order of example pairs. Due to our new example pair

order, only the synthesized rule shown in Fig. 36 is available when our second example pair is considered by our synthesis algorithm. Therefore, only one rule is matched with the second example pair, whereas, in the previous ordering, already two synthesized rules have been taken into account. This results in fewer matched objects and leads to a quite large rule.

In order to be independent from a special ordering of the example pairs, the synthesis algorithm is slightly more involved. In the extended version of the rule synthesis algorithm, not only already synthesized rules are matched against the newly extracted TGG-rule, but also the new TGG-rule is matched with all already available TGG-rules. If such a match is found, the algorithm handles this matching as in the case before.

In our example, after the synthesis of the third rule shown in Fig. 39, the synthesized rule will be matched against the second synthesized rule shown in Fig. 51. This results in two valid matchings and, as described before, the two successful matchings lead to a reduction of the second rule. Hence, the final result of the extended rule synthesis yields a set of TGG-rules which is nearly identical to the previously synthesized set of TGG-rules. The only difference between the synthesized rule sets is the numbering of the correspondence nodes. Remember that the numbering of the correspondence nodes reflects the order in which the TGG-rules have been synthesized. Since this order differs, also the numbering of the correspondence nodes is different – but this has no impact on the quality of the synthesized rules.

Altogether, due to the described extensions our rule synthesis algorithm is independent from the order of the given example pairs and what's more, it is also capable of processing a set of example pairs at once. However, an advantageous order of the provided example pairs increases the performance of our synthesis algorithm.

4.1.4 Extensions

Up to this point, our algorithm takes only objects and links of the given example pairs into account. Attributes, additional constraints, and reusable nodes have not been considered yet. In the following, we describe the synthesis support for these advanced concepts.

Attributes The provided example pairs can contain attributes with a particular value. In order to synthesize correct TGG-rules, we have to take care of these attributes too. In order to handle attributes automatically, the translation of a given example pair to the common TGG-formalism also considers the concrete attribute values of the involved objects. For example, the component elements of our project as well as places of the Petri net have an attribute *name*. As shown in Fig. 29, for a valid mapping these attributes have to be included in the TGG-rule as well.

Let us assume that a user provides an example pair relating a track component to a Petri net where both, the track component as well as the associated place, are named *t1*. As shown in Fig. 52, the translation will include the attribute *name* with an assigned string literal *t1* in both objects.

Figure 52: Translated example pair with attributes

In our approach, we use a simple heuristic based on string matching. Due to this heuristic, we can conclude that both attributes are related to each other, e.g., during a transformation from a project to a Petri net, a place has the same name as the corresponding component, and vice versa. As a consequence, we synthesize the attribute constraints from the example pair as shown in Fig. 53.

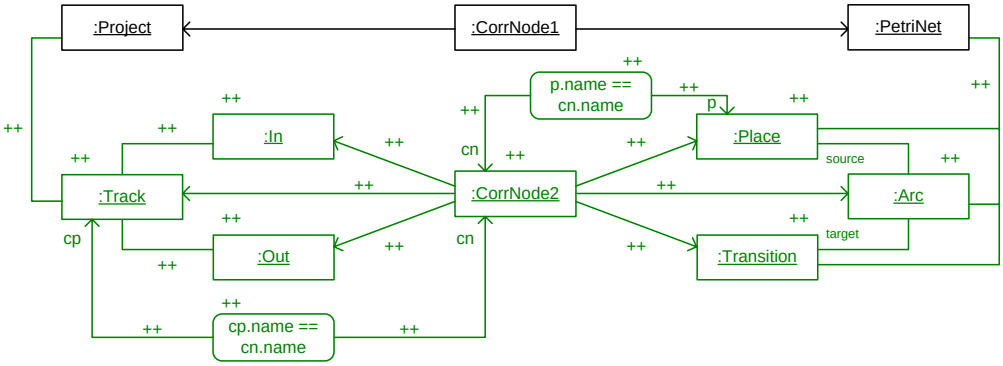

Figure 53: Synthesized rule with attributes

In the case that no such relation is found, we synthesize a simple constraint from the attribute value. For the sake of an argument, let us assume that a track component has a *kind* attribute in order to distinguish between a straight track and a curved track. Consequently, the values which can be assigned are the enumeration constants *STRAIGHT* and *CURVED*. Furthermore, let us assume that the user can create only example pairs where this attribute is set to a particular value. Therefore, he provides an example pair with the *kind* attribute set to *STRAIGHT*. The track component, however, is mapped in both cases to a transition, a place, and a connecting arc as in the previous example pair, i.e., it is mapped independently from the value of the *kind* attribute. During the rule synthesis, no related attribute is found. Therefore, the simple attribute constraint presented in Fig. 54 is extracted. This rule will take only straight tracks into account. In order to handle a curved track, the user has to provide a second example with the *kind* attribute set to *CURVED*.

In our example, there are just two different enumeration values. Therefore, it is feasible to provide a second example pair. However, if we imagine that an

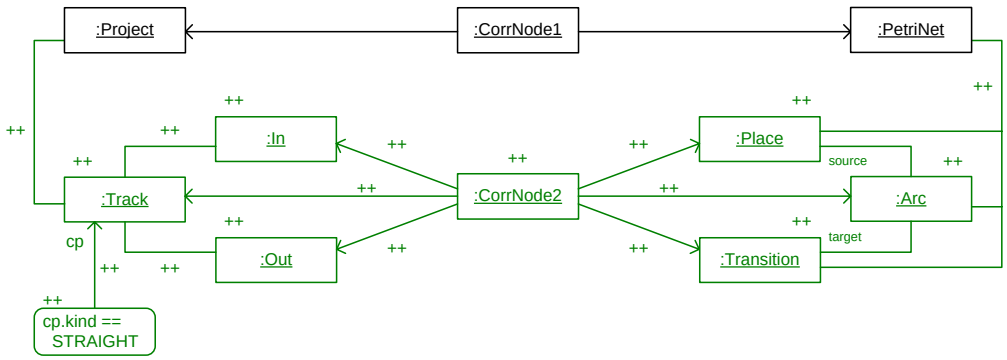

Figure 54: Synthesizing simple attribute constraints

enumeration can have many different values, things become much more complicated since for each value an own example pair has to be provided. Things become even worse, if more attributes in an example pair have to be considered. In such a case, for each combination of attribute values a separate example pair has to be specified.

At this point, our attribute synthesis and heuristic can be improved a lot. For example, the attribute synthesis could ignore attributes if at least two example pairs with the same object structure which only differ in the given attribute value have been provided. Or, concerning our heuristic, we could in addition search for some substrings in order to extract prefixes or suffixes for the attribute values.

For the moment, we deal with this kind of problems by user interaction, i.e., our synthesis algorithm makes initial proposals how to deal with the given attributes, but the final decision is left to the user. In that sense, our attribute synthesis algorithm is semi-automatic only. However, in most of our examples, the proposals are correct and have just to be confirmed.

Constraints As already described in Section 3.2, beside attribute constraints, there are also some more general constraints. In this section, we will show how such constraints are synthesized using our specification by example approach.

Let us consider once again the example from Section 3.2. There, we introduced a rule which transforms a train with a component into a token in the corresponding place of a Petri net. However, this transformation may be performed only if the corresponding place has no incoming arcs. For this purpose, we extended the rule in Fig. 30 with an additional constraint.

In order to synthesize a TGG-rule together with such a constraint, we have to provide – beside an adequate example pair for the mapping itself – an example expressing the constraint, i.e., a counter example describing the forbidden situation, which is also referred to as a negative application condition. For our example, this constraint is expressed in the graphical syntax of the Petri net by a place with an incoming arc. Although conceptually this would be sufficient, most Petri net editors do not allow to insert an arc without a source since this will lead to an inconsistency in the abstract syntax representation. For this

reason, we also include a transition in the example constraint, i.e., our graphical constraint comprises a transition, a place, and a connecting arc with the transition as source and the place as target. In Fig. 55, the translated example pair and the translated counter-example are shown.

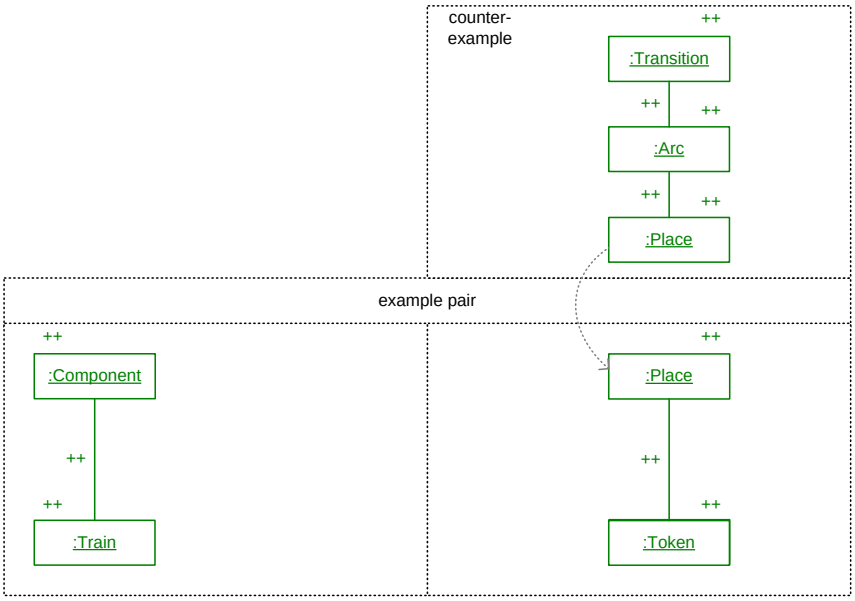

Figure 55: Translated example pair with counter-example

In order to extract a constraint, the synthesis algorithm tries to find a matching between the counter-example shown in the top of Fig. 55 and the example from the Petri net domain. Here, only the place object can be matched (depicted by the dotted line between both place objects). As a consequence, during the execution of the rule synthesis algorithm both objects are merged. The link to the unmatched object of the counter-example representing the arc is marked as a negative link and the arc object is included into the synthesized TGG-rule. Since the transition object can not be reached from the place without traversing the negative link, it is deleted from the synthesized rule. The result of the described procedure is presented in Fig. 56 using the short hand notation for negative links.

Reusable Nodes As described in section 3.3, there are some model elements that should neither be generated nor changed when applying a TGG-rule, but are only necessary since they are referenced from other model elements.

Typically, the referenced model elements belong to a separate domain like the component library. In order to handle these referenced model elements correctly and not to delete them from the TGG-rule during synthesis, these meta-model elements are annotated as reusable. During the rule synthesis, instances of annotated meta-model elements are not deleted from the TGG-rule at all. They remain within the TGG-rule regardless of whether they are connected to any newly created object or not.

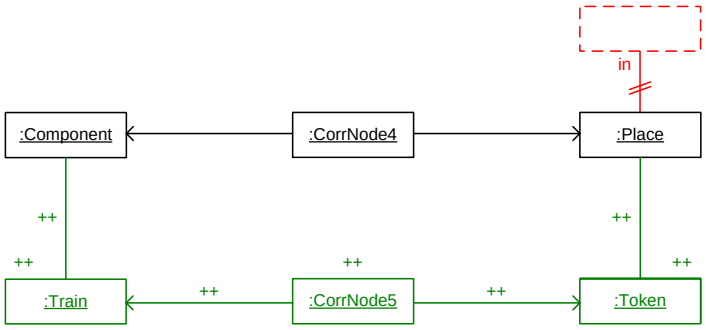

Figure 56: Synthesized TGG-rule with constraint

4.1.5 Recommendations

In the previous sections, we have presented an algorithm which synthesizes TGG-rules from simple correspondence example pairs. The example pairs are defined using the concrete syntax of the involved models. In the case of a visual modeling language the concrete syntax conforms to the graphical representation of the models. In that sense, the specification of the example pairs is performed in a more user-friendly way and no extra knowledge about the underlying TGG-formalism is needed.

The presented synthesis algorithm works fully automatically provided that no object attributes have to be considered. It is independent from a particular order of the given example pairs. However, one example pair is not sufficient in order to synthesize a universal set of TGG-rules that handles all model instances defined by the according meta-model of the modeling language. Rather, a set of example pairs has to be provided where each example pair has some similarities with at least one other example pair and where one additional concept of the modeling language is introduced. In the case that a particular concept is not covered in any of the given example pairs, the synthesized set of TGG-rules will not be able to handle this concept at all. On the other hand, from an example pair introducing more than one concept at once, a TGG-rule is synthesized that can only handle that concept in conjunction with the other concepts. Therefore, we recommend to start with quite small example pairs and to extend the examples step by step.

In order to support an incremental design process for the example pairs, the rule synthesis process works in an iterative manner. An overview of the supported process is shown in Fig. 57. The process starts with the definition of some example pairs. From the given example pairs, TGG-rules are synthesized. The synthesized TGG-rules can be validated, e.g., by executing a transformation on some test models and comparing the transformation result with the expected resp. required transformation result. In the case that the yielded model does not conform to the expected model, the set of example pairs can be further refined, modified, and extended until a final and valid mapping is achieved.

As mentioned before, providing semantically corresponding source and tar-

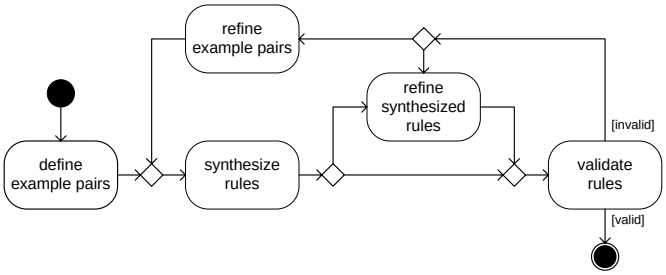

Figure 57: Process overview

get models, a validation of the synthesized TGG-rules can be accomplished either by transforming the source model and comparing the output of the transformation with the target model or by executing the integration scenario and checking whether both models are fully matched. Moreover, since corresponding model pairs have already been provided as example pairs for the rule synthesis, these example pairs can be also used for an automated validation of the TGG-rules. Of course, in that case only the correctness of the synthesis algorithm will be checked.

However, as already said before, our synthesis algorithm works only automatically if no attributes have to be considered. In the case that also attributes have to be considered, the algorithm works interactively, i.e., it proposes a solution for the attribute mappings but the final decision is left to the user. In addition, the number of needed example pairs depends on the number and range of the attribute values. For example, if an example pair contains two attributes where the first attribute can be assigned m different values and where the second attribute can be assigned n values, than altogether $m*n$ different configurations are possible. In the worst case, for each configuration a separate example pair is needed. Obviously, this increases the needed specification efforts and spoils the advantages of the specification by example approach.

In order to circumvent this problem, we allow the user to refine and modify the synthesized TGG-rules manually (cf. Fig. 57). This way, the attribute mapping can be generalized by hand, e.g., by providing a simple mapping function. Although the synthesis algorithm now becomes semi-automatic only, the main advantage is the reduced effort for the specification of the mappings. Now, the automated validation based on the provided example pairs also makes sense since not only the correctness of the synthesis algorithm but also the manually performed refinements are validated.

4.2 Specification in Graphical Syntax

One reason for TGG-rules having so many nodes is that we refer to their *abstract syntax* – i. e. the object diagrams – rather than to the graphical representations of these model elements. This becomes clear if we compare the informal definition of the relation of a track element to its Petri net in Fig. 9 on page 8 with the corresponding object diagram in Fig. 10. And it is even worse for the TGG-

rule for the splitting switch (top right in Fig. 9) to the corresponding TGG-rule in Fig. 12 on page 9. The main point is that, for most graphical notations, the underlying abstract syntax is much more verbose than the graphical syntax.

We could avoid much of this verbosity, if we could edit the TGG-rules directly in the graphical syntax of the two models – similar to the informal mapping of Fig. 9. The only parts that are missing there are the correspondence nodes. Figure 58 shows the completed example. In addition to the two parts of the two models, it shows the correspondence nodes (in some graphical notation) and how they relate the different objects in the graphical notation. The dashed part represents the canvas on which the model is drawn and the relation between the two overall models. This will be the left-hand side of the TGG-rule. Though, the correspondences make the graphics a bit more com-

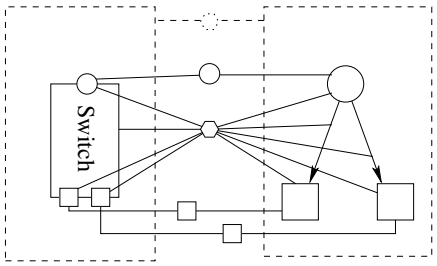

Figure 58: A TGG-rule on graphical syntax

plex, the representation of a TGG-rule is much simpler, and the relation to the actual parts of the model is more obvious than in the TGG-rule in abstract syntax. All other rules could be represented in a similar way.

Conceptually, this representation of TGG-rules in the graphical syntax of the respective models is not a big deal. The problem is in the implementation. But when we want to implement an editor for the TGG-rules in graphical syntax, things become more difficult – as we will see in a minute. First let us point out, that a TGG-rule editor in abstract syntax needs to know the meta-models of both models and the correspondence nodes; therefore, we can easily implement an editor for TGG-rules in abstract syntax. The problem with an editor for TGG-rules in graphical syntax is that we need to know how the graphical editors for both models are implemented and we need to integrate both of them into a single editor. Since every editor is implemented in a different way, it is hard – if not impossible – to implement such an editor in a generic way.

Recently, however, within the EMF Technology (EMFT) framework, the Graphical Modeling Framework (GMF) was proposed. This is pretty much a generic way to define a graphical editor on top of an EMF meta-model for some type of model. Basically, GMF needs to know how each meta-model object is graphically represented. With this information, GMF fully automatically generates the code for the editor. Since GMF editors are generated in a standard way, we could also generate a TGG editor joining two models – provided that both models have GMF editors.

We have not yet investigated this idea in detail and we did not yet implement

this idea. Still, implementing an editor for TGG-rules in the graphical syntax of the involved models is no longer out of reach – if we restrict ourself to EMF meta-models and GMF editors.

5 Realization

There are many different tools implementing TGGs and there are different editors for TGGs; here, we focus on the tools which perform the actual transformations, integrations and synchronizations with respect to some TGG-rules. An important implementation issue is the choice of an underlying technology, i. e., in which technology will the meta-models and the models be represented. Here, we discuss two examples: a tool based on the proprietary technology used in the *Fujaba Tool Suite* [43] and a tool based on the *Eclipse Modeling Framework* (*EMF*) [10].

In Fujaba, TGGs play an important role in the transformation and integration of different UML models, and therefore, Fujaba is a good choice when using TGGs in the context of model based software engineering. The EMF based tool might be better suited for more general kinds of applications. Actually, it is possible to implement adapters for all kinds of proprietary formats so that they can be used either within the EMF or the Fujaba technology.

We do not go into the technical details of these two technologies. Rather, we discuss the Fujaba and the EMF approach for another reason. Essentially both approaches need to do extensive pattern matching in order to apply rules. But, Fujaba *compiles* the TGG-rules into Java code for the pattern matching, i. e., it generates code for the transformations. The EMF tool interprets the TGG-rules while performing the transformations. These two approaches will be discussed in the following section.

5.1 Generative Approach

As already mentioned, in the generative approach, the TGG-rules are translated into code which is compiled and executed in order to perform the different application scenarios, i. e., model transformation, model integration, and model synchronization, with high performance. However, instead of generating the desired code from the TGG-rules directly, in a first step we derive simple graph transformation rules from the declarative TGG-rules and generate the code in a second step using Fujaba's built-in code generation facilities for graph transformation systems. This two-step approach is aligned to MDA's [33] approach in which a platform independent model is transformed into a platform specific model which is then used to generate the executable implementation.

Fig. 59 shows a simple graph transformation rule which transforms a connection from a project to an arc in a Petri net. The rule was derived from the TGG-rule presented in Fig. 50 by removing the ++ labels from all elements that belong to the project domain model. Note that although TGG-rules are bidirectional and no direction of a transformation is favored, a graph transformation rule transforming a model that is drawn on the left-hand side to a model drawn on the right-hand side is often called a *forward* rule.

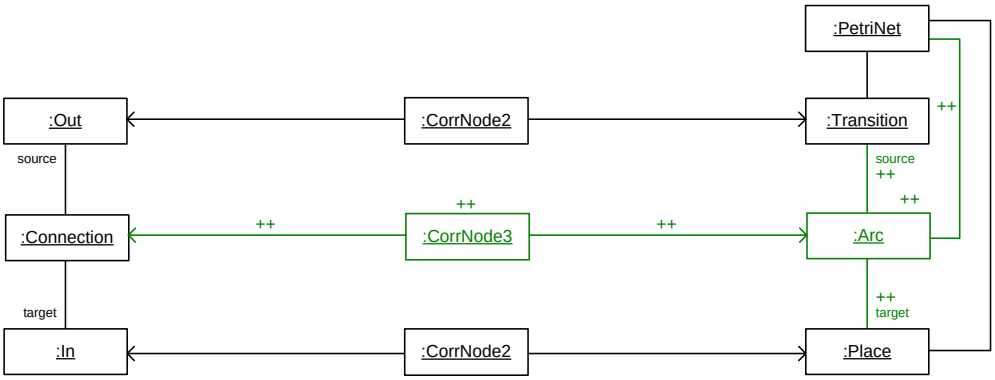

Figure 59: A forward graph transformation rule

In contrast to the forward rule, a graph transformation rule which handles the transformation from a model drawn on the right-hand side to a model drawn on the left-hand side, i. e., in our case from a Petri net to a project, is called *reverse* or *backward* rule. It is derived from the TGG-rule in the same way as the forward rule: we just remove the ++ labels from all elements that belong to the model domain rendered on the right-hand side, i. e., from the elements of the Petri net. It is presented in Fig. 60.

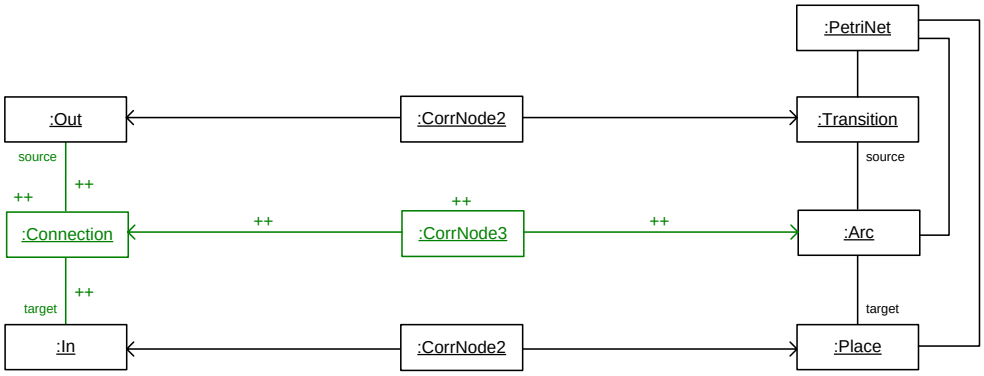

Figure 60: A reverse graph transformation rule

The third rule derived from the TGG-rule is responsible for the correspondence analysis between the involved models, i. e., for model integration. The rule is executed in the case that both models already exist and only the interrelation between the models has to be established, i. e. the rule creates the correspondence nodes and the traceability links where possible. The graph transformation rule for model integration is derived by removing all ++ labels from all elements that do not belong to the correspondence model. This graph transformation rule is called *relation* or *correspondence* rule and is shown in Fig. 61.

Up to now, the graph transformation rules derived can be applied as long as a valid matching is found. In particular, this means that we could apply a rule over and over again to the same matching. For example, a repeated application

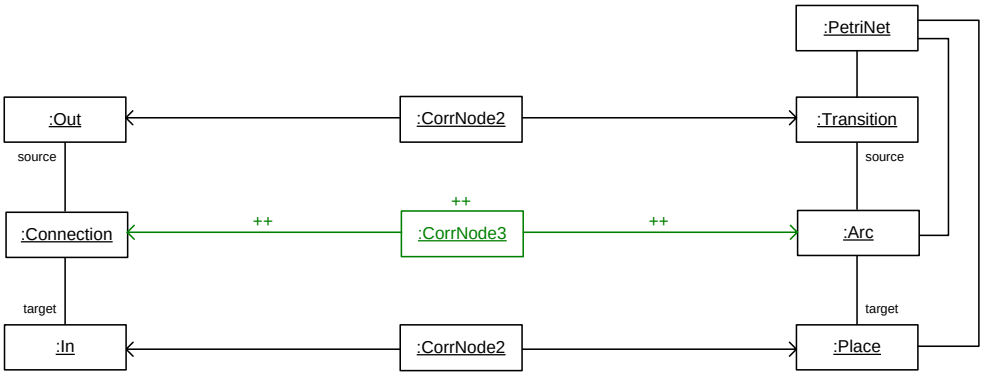

Figure 61: A correspondence graph transformation rule

of the forward rule from Fig. 59 to the same matching will introduce many arcs between the same transition and place. This neither makes sense nor does this reflect the intended behavior resp. semantics of TGGs.

In order to implement a correct behavior of the TGG-rules, we have to mark already mapped nodes within a transformation task. One alternative would be to add an attribute to the classes of the involved meta-models. However, this means that the meta-models have to be changed which is not always possible or wanted. In addition, a single attribute will not allow the model to be involved in different transformations. Another alternative would be to check for already assigned correspondence nodes. But, once again this solution will not allow the model to be involved in different transformations. Therefore, we have decided to realize the marking as a set of already handled nodes within a transformation task.

The derived forward graph transformation rule including the marking facility is shown in Fig. 62. The marking set is implemented using *handled* links. For example, in the forward rule from Fig. 62 it is checked whether the *Connection* node is not yet matched using a negative link between the *Task* node and the affected *Connection* node. If no such negative link exists, i. e., if the negative application condition evaluates to true, a valid match is found and the affected node is marked by creating such a link. In addition, the created *Arc* node is also marked as already handled. For a unidirectional transformation this will be not necessary. However, our transformation algorithm can be used for incremental transformations in both directions. For this reason, the handled nodes of the Petri net are also marked. The newly created *handled* links prevent matching these nodes another time by the same transformation task.

Another extension to the derived graph transformation rule are the links between the correspondence nodes. Due to the construction principle of our TGG-rules, each rule has at least one correspondence node which has to be already present for a successful application of the rule. This correspondence node is a necessary prerequisite for the application of the rule and therefore, the rule can be only applied if the required correspondence node was already created in a previous transformation step. Additionally, each successful application of a rule results in at least one additional correspondence node. As a consequence,

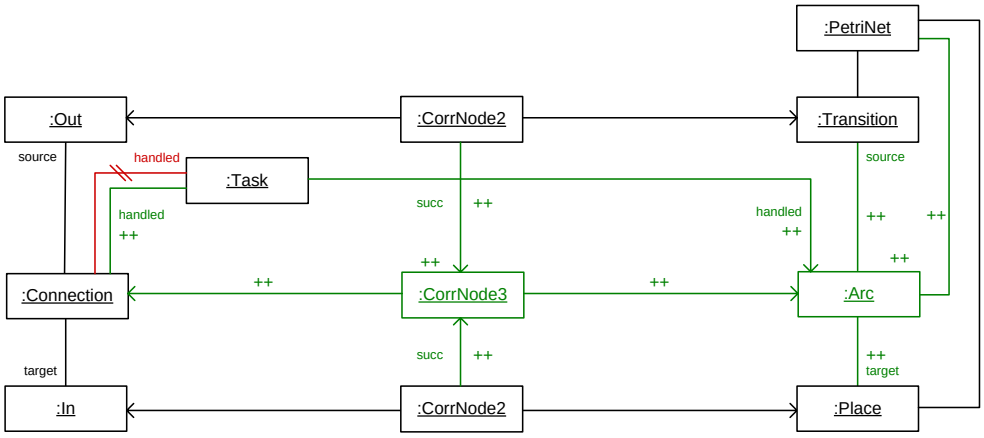

Figure 62: The extended forward graph transformation rule

the set of all TGG-rules implies a particular implicit execution order of the rules. This observation can be exploited to select only likely candidates for the rule application within an optimized transformation process such that we can apply a local searching strategy reducing the costs for the required pattern matching drastically. For this purpose, the execution order of the rules is stored and made explicit using the *succ* links.

In order to reduce the costs for pattern matching and in order to make our algorithm incremental we have to take the extended correspondence model with the *succ* links into account. With the additional links between the correspondence nodes, the correspondence model can be interpreted as a directed acyclic graph (DAG). It is a graph rather than a tree due to the fact that rules are allowed to have more than one correspondence node as a precondition (cf. Fig. 62). The graph is acyclic since in a rule application, we never connect already existing correspondence nodes by a link.

In Fig. 63 an overview of the realized incremental algorithm is shown. The presented algorithm comprises several activities which in turn are implemented using the aforementioned graph transformation rules. The combination of an activity diagram with embedded graph transformation rules is also called a story diagram [43]. The story diagram is derived automatically from a TGG-rule. In order to execute the derived story diagram, we automatically generate Java code from it using Fujaba's code generation facilities. Finally, this code is compiled to an executable format.

The incremental transformation and update algorithm traverses the correspondence nodes of the DAG using breath-first search. For each correspondence node the algorithm checks whether an inconsistent situation has occurred. This is done by retrieving an applied rule (find applied rule) and checking whether it still matches to the pattern structure (check pattern structure).

If the rule cannot be matched anymore, e.g., due to the deletion of a model element, we have found an inconsistency. In that case, the algorithm has to undo the applied transformation rule (undo rule application). This is achieved by deleting the correspondence node and all created elements. Note that by delet-

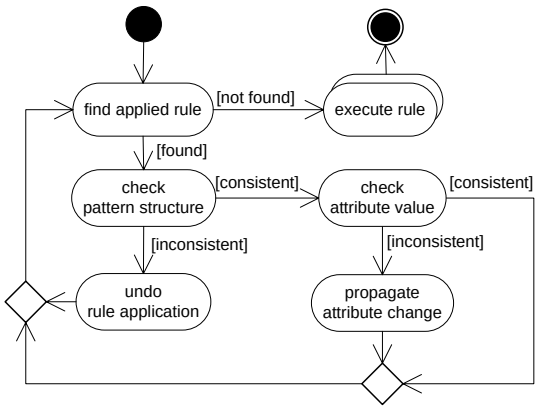

Figure 63: Incremental rule application

ing the correspondence node the precondition for all successors of the deleted correspondence node will not hold anymore. As a consequence, this leads both to the deletion of the succeeding correspondence nodes and the elements referenced by the deleted correspondence nodes.

In the case that the structure of the applied rule still holds and only an attribute constraint evaluates to false (check attribute value), it is sufficient to propagate the attribute value change in the current transformation direction (propagate attribute change). If all old rule applications have been checked, the algorithm searches for new model elements and transforms those elements according to the triple graph grammar specification (execute rule). Note that the transformation is executed as long as rules are applicable which is indicated by a for-each activity (execute rule).

The presented incremental algorithm can be used for unidirectional model transformations as well as for bidirectional model transformation and model integration. We can further optimize our incremental algorithm if the involved models support change notifications. In that case, the presented transformation algorithm starts to traverse the DAG at the correspondence node connected to the modified element and not at the root node.

In order to support incremental model synchronization enabling round-trip engineering between models, we have extended the presented algorithm by partial pattern matching and automatic completion facilities. This enables the reuse of nodes and allows us to create the missing parts only within a partially matched pattern. However, we are not going into details here and refer to [3] for an elaborate presentation of these extensions.

5.2 Interpreted Approach

As we have seen in the compiled approach, the basic task for transforming one model into another or for synchronizing two models is matching parts of the TGG-rules with the models and then create the remaining parts of the rule in the models.

5.2.1 Forward Transformation

We explain the basic idea of this mechanism for the forward transformation of one model into another. To this end, let us consider the models and the rules from Sect. 2.3 again where we transform a Component Tools project into a Petri net, where we start from the model as shown in Fig. 64.

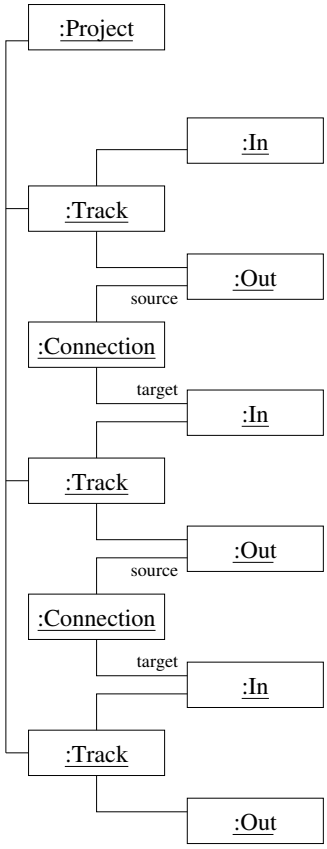

Figure 64: A project

For the transformation, we start with generating the start situation as shown in Fig. 65, which is the same situation as in Fig. 23. We only marked some nodes and links with an asterisk. This indicates that we have processed and mapped these elements already by a rule. In this case, we have used the axiom. Now, the transformation proceeds by finding a rule which maps to the existing nodes. For example, we can map the rule for *Tracks* as shown in Fig. 11. The black nodes and links (rules of the left-hand side of the rule) of this rule map to nodes and links which are marked with an asterisk already. The green nodes of the TGG-rule in domain *project* map to the corresponding nodes of the *project* which are not yet marked with an asterisk. The green nodes of the TGG-rule in domain *corresp* and *petrinet* cannot be mapped. This mapping is shown in Fig. 66.

Since all elements of domain project of the TGG-rule could be mapped to nodes of the project, the rule is applicable, and we can generate the nodes

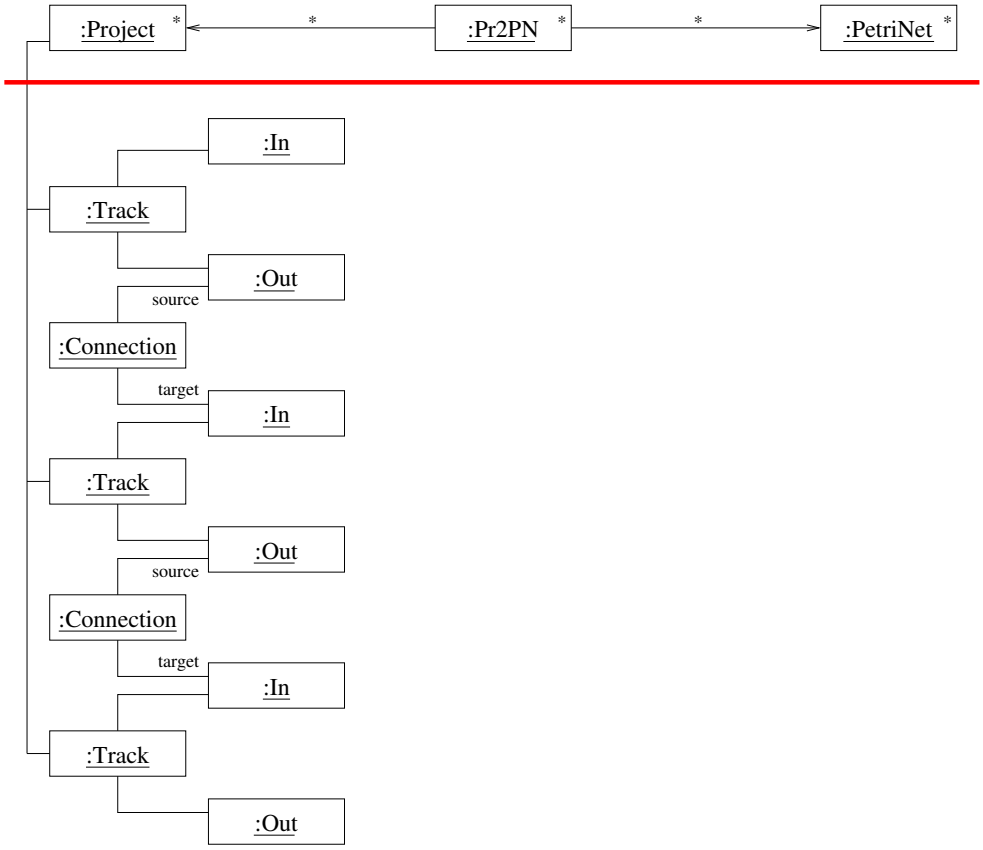

Figure 65: Start situation for transformation

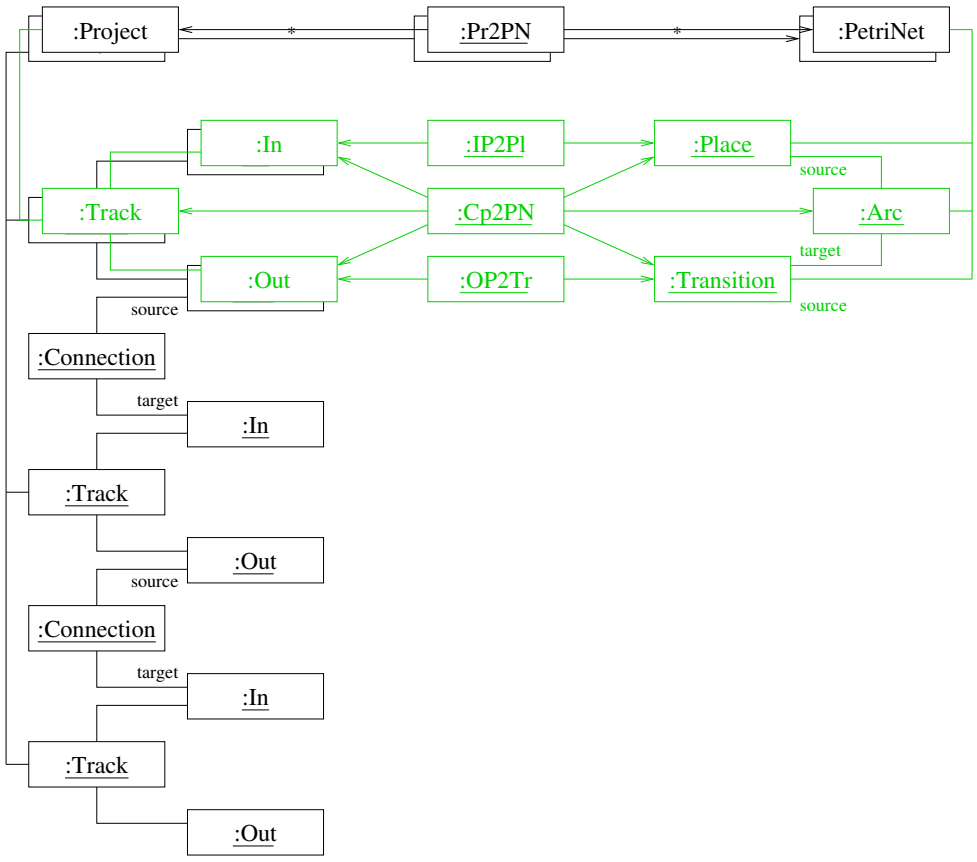

Figure 66: A matching TGG-rule

corresponding to the green nodes of the TGG-rule from domain *corresp* and *petrinet*. The result is shown in Fig. 67. Note that now, all nodes which where mapped to the TGG-rule are also marked with an asterisk.

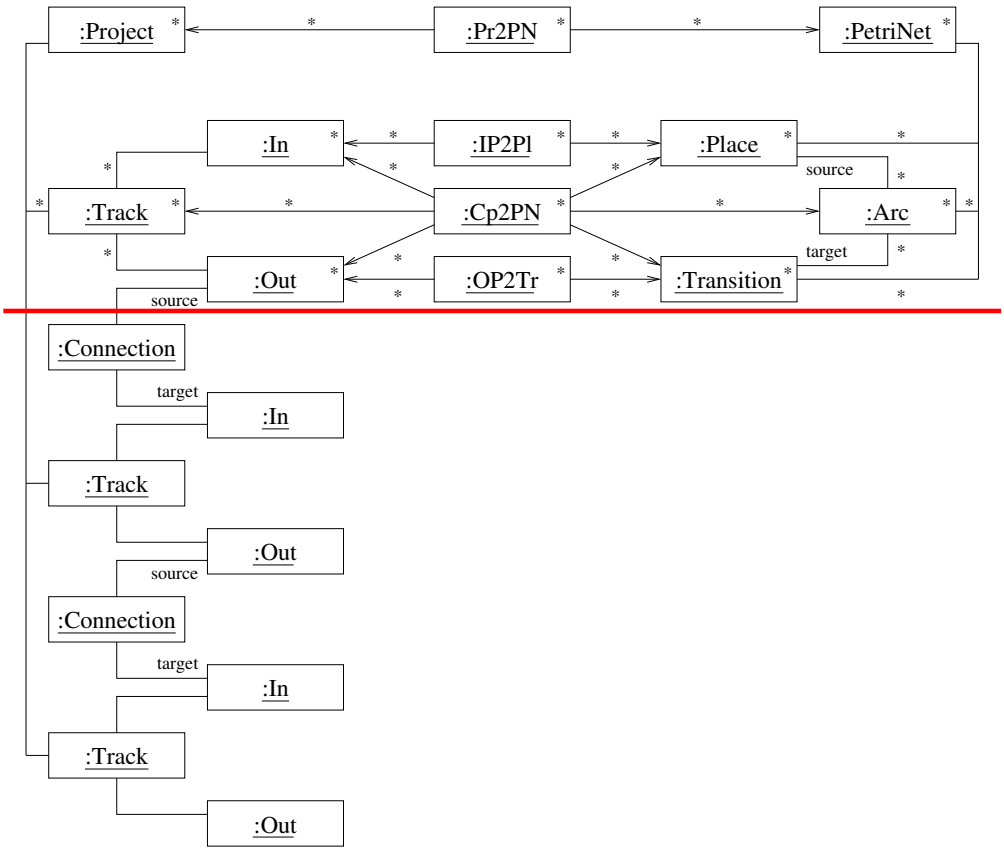

Figure 67: Application of the TGG-rule

Now, we could proceed matching the next TGG-rules. Since this proceeds along the lines discussed in Sect. 2.4, we do not show theses steps in detail again. The transformation ends, if we cannot match any TGG-rule anymore; the transformation was successful, if all nodes of the source model have been mapped, i. e., no unmarked node in the source model is left over. Moreover, we need to check all the constraint nodes that have been generated during the transformation process.

5.2.2 More Details

One important question in the interpreted approach is to find matching TGG-rules efficiently. Therefore, we need to know where to start searching for matching rules. To this end, we use the *front-line*. This is defined as the set of all links starting from an already mapped node (marked by an asterisk), but which are not yet mapped. In Fig. 65 and 67, the front-lines are also indicated by a red bold-faced line. When we match a TGG-rule, the links of the current *front-line* are the ones which are mapped to green links attached to a black

node of a TGG-rule. This is where we start to match a TGG-rule. Typically, there are not many different rules for such links so that we do not need to try many rules before finding a matching one.

Of course, there is the possibility that different rules match at the same links. In this case, there is non-determinism. There are different ways to deal with non-determinism. First of all, we can consider the constraint nodes for resolving the non-determinism and for selecting the TGG-rule with the best chance for a match. This might exclude some TGG-rules right away and, this way, resolve non-determinism and increase efficiency. Another way of dealing with non-determinism is backtracking; this, however, can be very inefficient. Therefore, in our applications, we designed the TGG-rules in such a way that non-determinism does not occur. But a theory – similar to a theory on deterministic push-down-automata and LR(k)-grammars in classical formal language theory – is still missing.

An important technical issue is marking of the already mapped nodes. One might be tempted to add a special attribute in the classes of the source and target model. But, this is not possible because we do not want to change the meta-models of the transformed models just for transformation purposes. Moreover, the same model might be involved in several transformations, so that a single attribute would not be enough. Therefore, the marked elements of a model are maintained as a set in the transformation engine. Moreover, for mapping a rule to a model, we need to navigate trough the model; often it is convenient to navigate links backwards even if the link is directed. To this end, we use the EMF cross referencing mechanism, which allows the interpreter to navigate links in both ways.

5.2.3 Other Scenarios

For the backward-transformation and the integration scenario, the interpreted approach for TGGs works in the very same way as explained in the forward-transformation. The only difference is, which domains need to be matched to the existing models and which are generated. The synchronization scenario is a bit more involved. Since the matching can be done in all domains; and the rest will be generated. The question is what does it mean to match sufficiently in order to generate the rest. This clearly depends on the field of application and there needs to be some *metric* which reflects this sufficient matching. The matching algorithm itself, however, is the same as for the other scenarios. In some cases, it might happen that an earlier matching is destroyed since in one or both models some nodes have been deleted. In this case, it might be better to not generate the missing elements, but to delete all the other elements of that rule from both models. Again, it depends on the field of application and there needs to be a metric for deciding to either delete the remaining elements or whether to add the missing elements.

6 Tool Support

As outlined in the previous section, there exist two different approaches for the execution of the TGGs. In this section, we present the implemented tool support for both approaches. The tool support for the generative approach is presented in the first subsection. In the second section, the tool support for the interpreted approach is presented. In the last section, we present a technique which allows both tools to interact with models that are based on different technologies.

6.1 TGG-Compiler

The TGG-compiler represents the generative approach which has been implemented in the Fujaba Tool Suite[9]. The available tool support includes an editor for the visual specification of the TGG-rules, a component for the automatic extraction of the graph transformation rules, and an engine for the incremental execution of these rules.

For the visual specification of TGG-rules we use the TGGEditor presented in Fig. 68. This editor is implemented as a Fujaba plug-in and ensures conformance to the source, the correspondence, and the target meta-models. For this purpose, the required meta-models have to be specified in Fujaba as class diagrams. In addition to the editor, there is also a plug-in implementing the specification by example approach. This plug-in enables the specification of example pairs from which it synthesizes TGG-rules. The automatically synthesized rules can be further refined using the editor.

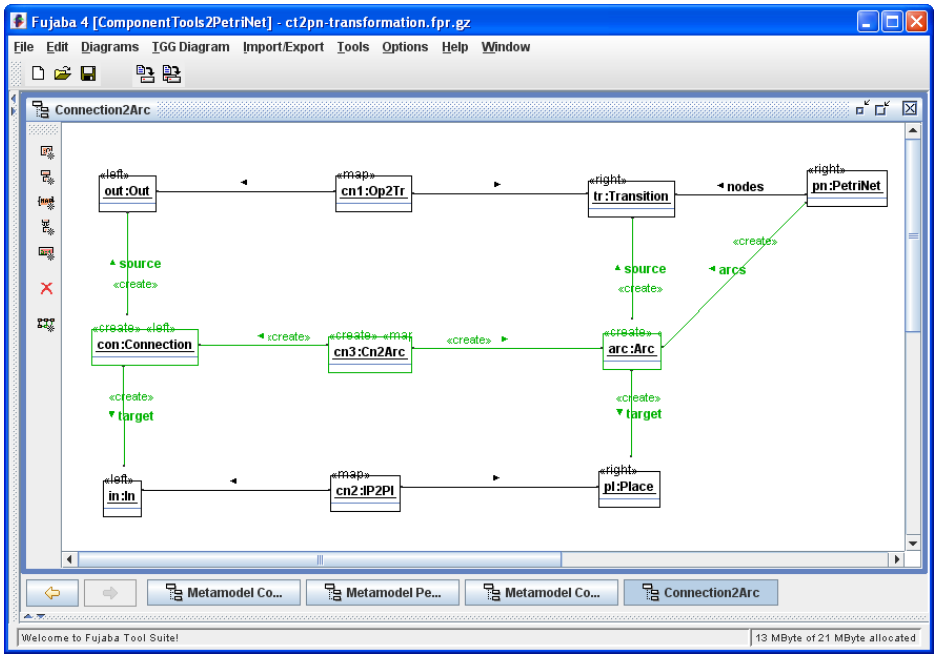

Figure 68: TGG editor

[9] see www.fujaba.de

In order to execute the specified TGG-rules, we derive from each TGG-rule corresponding graph transformation rules. From these automatically derived graph transformation rules, we generate Java code using Fujaba's code generation facilities. In order to execute the transformation, we compile the generated code and bundle it into a single JAR archive file. In addition, the JAR file has to provide a configuration file with listed rules. The archive represents the catalog of rules defining the TGG. Once the catalog is available, model transformations, model integrations, and model synchronizations can be carried out.

The first step to execute one of the scenarios is to setup an appropriate task. For this purpose, we have to name the task and select the catalog containing the compiled TGG-rules. Thereafter, we have to select the source and/or the target model. For example, in a transformation scenario, we can select one domain as a source model and transform it initially to a target model of the other domain. Or, we can select a model from the other domain and transform it in the reverse direction. If we select two models, both models are checked for corresponding parts and the related parts are connected by correspondence nodes. This corresponds to the model integration scenario. For synchronization purposes, the different tasks can be re-executed each time a model changes.

The execution of the TGG-rules is performed by the two Fujaba plug-ins MoTE and MoRTEn. MoTE is the abbreviation for Model Transformation Engine. It is the core library for the execution of TGGs and can be also used without Fujaba. MoRTEn is the abbreviation for Model Round Trip Engineering. MoRTEn integrates the MoTE library into Fujaba and provides a graphical user interface to setup and control the execution tasks.

MoRTEn is intended to be used to test a specified model mapping during development. After a mapping is specified and tested, it can be integrated into any Java-based software tool with Fujaba-compliant meta-models or appropriate model adapters (cf. section 6.3) using the MoTE library. An example for such an integration is presented in [17] where Matlab/Simulink models are automatically transformed to pattern specifications by utilizing the MoTE plug-in.

6.2 TGG-Interpreter

In contrast to the generative approach, the interpreted approach is implemented using the emerging technology of the Eclipse platform [9]. In particular, the tool support comprises a visual editor for the specification of TGG-rules and an interpreter engine in order to execute the specified TGG-rules. Both tools are based on the Eclipse Modeling Framework (EMF) [10].

The reason for using EMF as an underlying technology is that this technology is widely adopted and many modeling tools are using EMF as a basis for their own meta-model implementations. As will be outlined later in section 6.3, this is indeed not a hard requirement for the specification and application of TGG-rules, but it is quite advantageous since the model instances can be accessed directly and in a quite uniform way.

Another reason for using the EMF technology is the fact that EMF is supported by many other projects and tools. For example, the Graphical Modeling Framework (GMF) [11] enables to generate fancy graphical editors from

EMF meta-models almost automatically. This facility was used to generate a graphical TGG-editor for the interpreted approach. The generated editor was enhanced by further features increasing the usability of the editor. For example, when selecting a particular node, the editor proposes to create further nodes that are potentially reachable from the selected node. This eases the specification of TGG-rules and makes it more comfortable. A screenshot of the generated editor is shown in Fig. 69. In this editor, there are no swimlanes for the representation of the different domains. Rather, the domains are represented as special *domain* nodes. The membership of a node to a particular domain is established by a link between the node and a domain node (cf. domain nodes on the bottom).

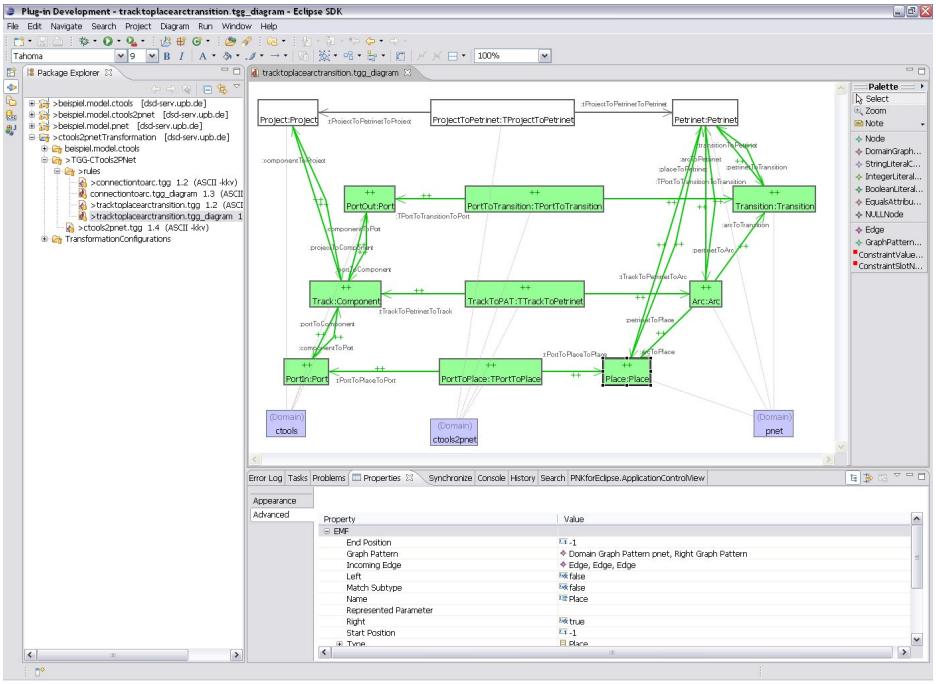

Figure 69: The generated TGG-editor for the interpreted approach

The TGG-rules specified within the editor are used by the TGG-interpreter engine to perform transformations of EMF instance models. For this purpose, the specified TGG-rules are stored using EMF's default XML persistence facilities. In addition, the interpreter has to be configured step by step using a configuration wizard. A configuration provides details in order to start a transformation. The configuration consists of the XML-based rule definitions stored by the editor and a start context with a source and target domain. After a valid start configuration is provided, the transformation can be executed.

The TGG-interpreter supports the transformation of a source model into a target model, reusable nodes, a basic set of attribute constraints, i.e., string, integer, and boolean literal attribute values as well as simple attribute constraints specifying the equality of two attribute values. The TGG-interpreter is still under development and therefore, model integration and model synchro-

nization are not supported yet. But it should be easy to extend them. Also, an integration of OCL for the specification of more expressive constraints is not yet completed. However, these additional capabilities are under development and will be provided in future releases.

6.3 Tool Adapter

In the previous two sections, we have presented the implemented tool support for the generative and the interpreted approach which are both based on different technologies. In the generative approach, i.e., the TGG-compiler, it is necessary that the meta-models are implemented according to the rules of Fujaba. In the interpreted approach, i.e., the TGG-interpreter, it is necessary that the meta-models are implemented according to the rules of EMF. The compliance with these requirements allows navigating between model elements, accessing and modifying model elements, as well as creating new model elements. These kind of requirements are fundamental for the implemented algorithms in both tools – if a meta-model implementation does not meet these requirements, the underlying algorithms will not work with that model at all.

However, there are modeling tools that do not meet this requirement and do not provide a Fujaba or EMF compliant implementation of their meta-models. Of course, the tool's meta-model implementations can not be changed. In addition, instead of following some other standard implementation for the access of their models, the tools often do not publish their underlying meta-models but rather provide a rudimentary Application Programing Interface (API) for this purpose only. In the worst case, i.e., if the meta-models are not available, a conceptual meta-model has to be reverse engineered from the API.

In order to allow our algorithms to interact with such models anyway, we propose to use the adapter design pattern [14]. The adapter design pattern converts an interface of a class into an interface which other clients expect, i.e., the adapters let the interpreter resp. compiler work together with the models of the proprietary tools.

In Fig. 70, an overview of the proposed tool adapter approach is presented. In the upper part of Fig. 70, the relations between the TGG-rules and the involved meta-models are shown. The specified TGG-rules are utilized by a TGG-engine, which could be either an interpretative engine or an engine executing the compiled TGG-rules. In the bottom part of Fig. 70, a transformation scenario is shown. In particular, the TGG-engine transforms model A to model B which are concrete instances of their corresponding meta-models.

In the presented transformation scenario in Fig. 70, the meta-model A has a compliant implementation that fulfills the requirement of the used TGG-engine. Therefore, the TGG-engine is able to access model A directly. In contrast to that, the meta-model B has only a proprietary implementation. In order to perform the transformation anyway, an adapter meta-model B' with an engine compliant implementation is introduced. Due to this intermediate adapter implementation, the engine modifies model B indirectly by performing all operations on the compliant adapter model B' which in turn delegates the access operations to the proprietary model B.

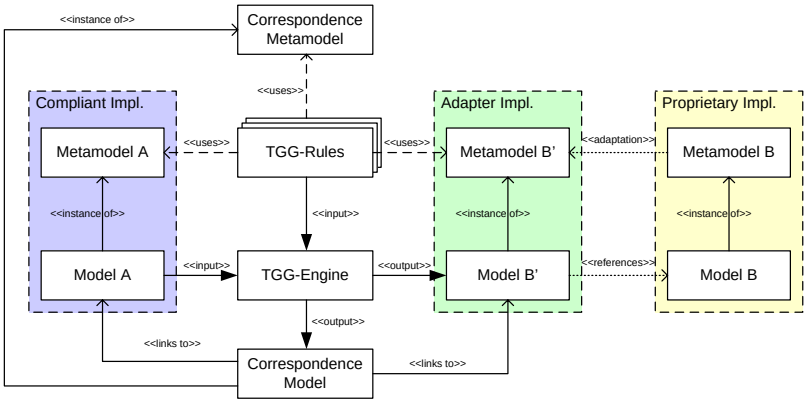

Figure 70: Overview of the tool adapter integration

In order to implement a compliant model adapter, either Fujaba's or EMF's meta-model facilities can be used in order to specify the involved meta-model. From this meta-model specification, the implementation can be derived automatically using the built-in code generation in Fujaba or, respectively, EMF. Up to this point, the automatically derived meta-model implementation does not provide any access to the proprietary meta-model. In order to establish such a linkage to the proprietary meta-model, we have to extend and modify the generated meta-model implementation manually using the adapter pattern. For this purpose, we remove all attributes from the generated code and replace the generated access method implementations using the available API for the proprietary meta-model. Note that we replace the method implementations without changing the method signatures in order to preserve the desired interface for the TGG-engine.

Following the described process, we have implemented model adapters in several projects, e.g., for an automata meta-model and a Matlab/Simulink meta-model [17]. The resulting model adapter implementations are stateless with lazy object initialization, i.e., the adapter objects are created only on demand. Additionally, the realized adapters keep a list of already adapted model elements and reuse them each time the model element is revisited. Thus, a model element is always represented by the same adapter object and adapter object identities are preserved. This guarantees a fast access to already adapted model elements and reduces at the same time the number of needed adapter objects. Of course, in the worst case, i.e., if all model elements have to be examined, this will result in one adapter object for each adapted model element anyway. Here, some further optimizations concerning the adapter implementations are possible.

The implementation of a model adapter is a convenient way in order to bridge the technology gaps between different tools and their meta-models. The implementation of an adapter can be further automated, if the meta-models that are to be adapted conform to some standard or implementation guidelines. For example, adapters for meta-models that are based on the Java Metadata Interface (JMI) can be generated fully automatically from their specification.

7 Related Work

In this section, we give an overview of the related work concerning model transformation, model integration and model synchronization. In addition, we discuss approaches which ease the specification of mappings resp. transformation rules in order to increase the usability.

7.1 Model Transformation

Motivated by OMG's Request for Proposal (RFP) on Query/Views/Transformations (QVT) [34], model transformations have been put into the focus of many research activities. Meanwhile, a first version of the Final Adopted Specification [35] is published. In this specification, incremental model transformations are an important issue. However, up to now there are just two implementations for the operational part of QVT [7, 13]. These implementations do not support incremental model transformations for synchronization purposes. A prototype supporting incremental model transformations is the Model Transformation Framework (MTF) [20] developed by IBM. The prototype implementation of the declarative part of QVT is able to synchronize node addition and removal, however, MTF lacks support for defining custom constraints. Unfortunately, so far, a full synchronization is therefore not possible [24]. In addition, there is no performance data nor any publication describing the used approach available.

However, beside QVT there is a large number of approaches for model transformation – each for a special purpose and within a particular domain with its own requirements. Here, we can only give a brief overview and refer to [8] for a survey.

A well-known approach for model transformation is XSLT [48]. It is used for the transformation of models represented as XML documents. Hence, a model has to be exported to this representation and is then transformed in one step. Incremental change propagation is therefore not supported. In addition, the model transformation has to be expressed using the concepts offered by XML and XSLT which is quite verbose and hard to read.

Another class of transformation approaches comprises visual transformation languages which are based on the theoretical work on graph grammars and graph transformations. These approaches interpret the models as graphs and the transformation is executed by using graph rewriting techniques. Examples for model transformation approaches based on graph grammars and graph transformation include VIATRA [45] and GReAT [47].

VIATRA is a framework for the definition and implementation of transformations in the context of transformation-based verification and validation. In this framework, rules are specified visually using the UML notation. From this specification unidirectional model transformers are derived. Similar to the XSLT approach, the input models for the transformers have to be exported to an XMI format. The result of the transformation is once again an XMI document. In-memory model transformations as described in [27], bidirectional transformations, traceability, and consistency maintenance between the transformed

models are not supported. However, in [46] some initial ideas on incremental graph transformations for increasing the performance of the transformation mechanism have been presented.

The GReAT model transformation system offers an operational specification technique based on graph rewriting for complex domain-specific model transformations. It offers language features for an explicit control flow with input and output parameters for passing objects to the transformation rules. However, GReAT supports only unidirectional transformations in a batch-oriented way. No further traceability information about the current transformation is provided. This prevents both incremental transformations and any consistency maintaining activities between the models after an applied transformation.

A feature shared by all mentioned approaches is that the transformation must be specified for each transformation direction separately. Hence, these approaches are not well suited for the specification of bidirectional transformations. A bidirectional transformation approach is BOTL [32]. It offers a UML-like notation for rule specification comparable to graph transformations. Like the two other approaches, the transformation is batch-oriented and not incremental.

The discussed graph grammar based approaches do not provide any explicit traceability information about the model transformation. This prevents both incremental transformations and consistency maintaining activities for model synchronization after an applied transformation. In contrast to that, triple graph grammars are a particular technique tuned to the specification and execution of incremental transformations in both directions.

Triple graph grammars were motivated by integration problems between different tools where interrelated documents have to be kept consistent with each other [4, 29, 30]. In this field, triple graph grammars are used for the maintenance of the required traceability links between different document artifacts. In [4] the transformation algorithm operates interactively. In contrast to our approach, the transformation algorithm therefore relies heavily on the guidance of the user which is not practical if very large models have to be transformed. The incremental transformation approach in [21] is triggered by user actions like creating, editing, or deleting elements. However, this requires the specification of all possible user actions and appropriate activities for the updates. Although the specification is done visually using graph grammar based techniques and is therefore much more comfortable than ad-hoc programming, the required specification effort increases with the number and granularity of the available user actions. Due to this operational characteristic, the overall consistency of the approach is difficult to guarantee. In addition, a complete model transformation from scratch is not supported whereas our approach handles both cases. However, some of the work served as a starting point for our approach. In particular, we rely on the proposed attribute update propagation techniques [4, 29] and the correspondence dependency introduced by [30].

7.2 Model Integration

Model integration means different things to different people. For example, the integration of models is often seen as a process where a single model is created from two or more models. In the domain of model management, this kind of integration is also referred to as model merging [5, 42]. In distinction, model integration as presented in our approach does not yield a new single model in one particular formalism. Instead, it leaves the given models in their own formalisms and establishes a correspondence mapping between them only. Once again, in the domain of model management, this kind of model integration is defined by a match operation: given two models as input a mapping between both models is calculated and returned as output [5, 42].

The closest related work in the field of model integration similar to our understanding is the work on tool integration in [4, 30]. This is quite obvious, because the performed tool integration is heavily based on the integration of the produced documents which can be seen as models, too. The document integration in turn is achieved using the TGG-approach.

A quite prominent tool for defining mappings between two models is the ATLAS Model Weaver (AMW) [12]. The main idea behind model weaving is to support modelers to establish links between individual model elements of models or meta-models. These links represent a mapping between the models resp. meta-models and can serve as input for further operations. Model weaving can be performed manually by the user [12] or automatically through some user-defined mapping operations or heuristics [12, 39]. In contrast to that, in our approach the mapping is specified explicitly and desired operations for checking and creating the relationships are derived from this specification automatically. In addition, in our approach, the specification of such mappings can be done in the concrete syntax of the models, whereas the relationships resp. links in [12] have to be defined on the abstract syntax representation of the models.

7.3 Model Synchronization

A classification of the model synchronization problem and an accompanying synchronization approach between a feature model and its specializations is given by Hwan et al. in [24]. In their work, the synchronization approach is based on traceability links between the interrelated models. These links are introduced during the generation of an initial specialization by cloning the original feature model. After the traceability links are constructed, they are used to propagate changes made in the feature model to the corresponding specialization models, i.e., the model synchronization works in one direction only. In addition, due to the nature of the presented model synchronization problem, the introduced traceability links represent one-to-many relationships. As the authors admit, this kind of synchronization is usually easier to implement than those involving many-to-many relationships. In contrast to that, our synchronization approach is capable of handling many-to-many relationships between elements of models in different notations as well as in different directions.

Ivkovic and Kontogiannis developed an approach for model synchronization

which is based on implicit traceability relations, i.e., the relations are defined and encoded between the meta-models rather the interrelated models [26]. This prevents fine-grained relations between models and restricts the approach significantly. Moreover, in their approach, special graphs for the purpose of model synchronization have to be derived from the meta-models. In addition, respective modifications represented as atomic graph operations on nodes and edges like insert, delete, modify, etc., have to be specified and implemented. In order to synchronize two models, the operations applied to the source model are traced and transformed to corresponding operations for the target model. Then, the transformed operations are executed on the target model. In a final step, an equivalence relation checks whether the synchronization was executed successfully. For the definition of a model synchronization between two models seven steps have to be accomplished. This is too complex if customizations of the model synchronization should be allowed to end users. Moreover, the authors agree that in practice implicit model synchronization will not suit all synchronization scenarios.

Hearnden et al. extend a declarative logic-based transformation engine in order to incrementally synchronize a target model with source model changes [22]. The presented approach records a transformation execution and maps changes in the source model to the execution record. This enables the calculation of necessary updates of the target model in order to keep both models consistent to each other. The solution comes at the cost of a permanently maintained transformation execution context. For large transformations, further optimizations of the extra needed space for the execution context have to be considered. In addition, it is not clear whether a bidirectional synchronization can be executed on the same execution record or if one execution record for each transformation direction is needed. However, utilizing model transformations for synchronization purposes is quite obvious and seems to be a promising approach.

Another set of related work deals with the synchronization of a model with its code. There are many tool environments which generate a code skeleton out of class diagrams, or which retrieve a class diagram from code and try to keep them consistent to each other [7, 25, 37]. However, in most current tools this synchronization is achieved in an ad-hoc and hard-coded manner. For this purpose, often changes to one common model are allowed only and both, the code and the model, are seen as special views on that model. The synchronization is achieved by utilizing some concepts like the Model-View-Controller paradigm [38] to update the views if the model changes. However, this approach does not allow to adapt the mapping between the code and the model to user or enterprise specific needs.

Other approaches for automatic synchronization propose to use a combination of forward and reverse engineering techniques that is also referred to as round-trip engineering [1, 23]. The main drawback of this approach is that for a given forward engineering function an inverse function for reverse engineering has to be derived. This is only possible if the model-to-code mappings are bijective. However, this is rarely the case [41]. In addition, this approach does not work anymore if a developer refines the implementation by completing

the generated skeleton code and makes thereafter some changes to the model – the forward engineering step will overwrite the manual modifications. In a round-trip scenario, the manual modifications should be preserved.

7.4 Usability

In most visual approaches for model transformation, e.g., [32, 45, 47], model transformations are specified over the abstract syntax of the source and target languages described by the corresponding meta-models. Although such a visual notation for model transformations has the advantage of representing the patterns of the source and target models in one diagram, the drawback of this approach is that the meta-model based representation becomes quite verbose for non-trivial model transformations. In order to prevent scattered specifications, Bettin [6] presents some ideas for a more compact representation of meta-models. Based on this concise meta-model representation, a possible notation for the specification of a model transformation is proposed. However, the paper gives only some rough ideas and a quite simple example in order to motivate the usage of a more compact visual syntax for model transformation. Since more elaborate examples are missing and no practical experience with the proposed approach has been made, it is arguable if the approach will be useful in practice at all.

Baar and Whittle investigated how model transformations rules can be made much more compact and easier to read by using the graphical syntax of the involved modeling languages [2]. Their approach conforms to our idea of specifying TGG-rules in the graphical syntax of the modeling languages. However, in contrast to our idea where the graphical editor for the transformation rules is generated automatically from the graphical editors of the modeling languages, in [2] the meta-models and the graphical rendering of the involved languages have to be adapted to the imposed requirements manually. In addition, further graphical objects, e.g., for the representation of abstract classes, have to be defined. As the authors admit, the adaption of the meta-models and the existing rendering is quite tricky and represents the main bottleneck of their approach.

Wimmer et al. propose a by-example approach for the definition of correspondence mappings between models in the graphical syntax of the modeling languages and to derive the model transformation rules from this definitions almost automatically [49]. For this purpose, in a first step, the user has to provide one or more semantically corresponding models covering the desired concepts of the modeling languages. In a second step, the user has to define the correspondence mappings between the elements of the involved models. From this definition, and an additional mapping between the abstract and concrete syntax which has to provided as a prerequisite, in the third step, model transformation rules are derived. The presented by-example approach differs from our approach (cf. Sec. 4.1) in the way the examples have to be designed. In [49], one example covering all desired concepts of the involved modeling languages, a user defined correspondence mapping between the elements of both models, and an explicit mapping between the abstract and concrete syntax are needed in order to synthesize the model transformation rules. In our approach, differ-

ent example pairs with some similarities and differences have to be provided. Although the authors also recommend to provide more but smaller examples in their approach, the additional user-defined mappings between model elements are still required. In contrast to [49], in our approach the example pairs are sufficient. In addition, we do not need an explicit mapping between the abstract and concrete syntax. The authors also remark that their approach does not allow a fully automated transformation rule derivation due to the fact that some ambiguities can be only resolved by user interaction. In addition, since no tool support is available yet, it is not clear if the presented ideas also apply to more elaborated transformation examples than the presented sample transformation.

Another by-example approach for the definition of model transformation rules was introduced by Varró in [44]. The overall aim of the approach is comparable to ours, but the concepts differ from each other. For example, the mappings are defined in the abstract syntax of the modeling languages whereas our approach uses the concrete syntax. In addition, in the approach presented in [44], a prototypical mapping has to be established using reference nodes in order to describe critical cases of the model transformation. These reference nodes allow to define a one-to-one mapping only. In order to generate the model transformation rues from the prototypical mapping, additional user interactions are required. For example, further reference nodes have to be added. Or, in order to reduce the number of generated rules, the user has to refine the generated rules manually. Hence, in contrast to our approach, the transformation rules in [44] have to be developed iteratively, whereas in our approach the iterative development is rather an option but not a requirement.

8 Conclusion and Future Work

In this paper, we have discussed TGGs as a technology for model transformation, integration, and synchronization. The focus of this paper is the spirit of TGGs and on how extensions can be made in a way compatible to their spirit. Moreover, we have discussed different ways of implementing TGGs and tools based on these ideas. Some of the extensions proposed in this paper, however, still need to be implemented in the tools.

Still, there are many interesting open questions – theoretical as well as practical ones.

- A theory for making TGGs deterministic or – at least – a sufficient condition when the transformations are deterministic is still missing (see Sect. 5).

- TGGs are very well suited for keeping track of relations among models that have a similar structure. For models with a quite different structure, TGGs are not so well-suited. We believe that TGGs could be combined with other approaches – such as templates, other operational approaches, or hybrid transformation languages – in order to be more flexible. This, however, needs more research.

- For the synchronization scenario, we need to use metrics or heuristics that describe how many elements of a rule need to be mapped in order to generate the remaining parts. The concrete metrics or heuristics depend on the field of application. This needs further investigation. Moreover, there needs to be a mechanism in order to incorporate different metrics into a transformation resp. synchronization engine.

- TGGs define the relation between two models in a very local way. This is close to the definition of a structural operational semantics (SOS). Therefore, this kind of transformation is well-suited for verifying the semantical correctness of transformations. This also needs further investigation. Some initial work in this direction can be found in [16, 31].

- Up to now, TGG-rules are quite large compared to their counterpart in graphical syntax. The reason is that the TGG-rules need to be formulated in abstract syntax. As discussed in this paper, it is possible to implement a framework on top of the TGG-engine and EMF, which edits the TGG-rules in graphical syntax. The implementation of such a tool, however, is still missing.

QVT relational is an other transformation technology, which has a similar characteristics as TGGs. Actually, some of the extensions discussed in this paper have been inspired by QVT. And it turned out that QVT core can be implemented by the help of TGGs [18, 19]. This shows that TGGs cover the basic concepts of *relational transformation technologies*.

References

[1] U. Aßmann. Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science*, 82(5):1–9, Apr. 2003.

[2] T. Baar and J. Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *Sixth International Andrei Ershov Memorial Conference, Perspectives of System Informatics (PSI)*, LNCS, pages 84–97, 2006.

[3] J. Backsmeier. Inkrementelle Modellsynchronisation mit Tripel-Graph-Grammatiken. Master's thesis, Department of Computer Science, University of Paderborn, Nov. 2006.

[4] S. Becker, S. Lohmann, and B. Westfechtel. Rule execution in graph-based incremental interactive integration tools. In *Proc. Intl. Conf. on Graph Transformations (ICGT 2004)*, volume 3256 of *LNCS*, pages 22–38, 2004.

[5] P. A. Bernstein, A. Y. Halevy, and R. A. Pottinger. A vision for management of complex models. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(4):55–63, 2000.

[6] J. Bettin. Ideas for a concrete visual syntax for model-to-model transformation. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[7] Borland. *Together Architect*, 2006. http://www.borland.com/us/products/together.

[8] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Oct. 2003.

[9] The Eclipse Foundation. *Eclipse*, 2007. http://www.eclipse.org.

[10] The Eclipse Foundation. *Eclipse Modeling Framework (EMF)*, 2007. http://www.eclipse.org/emf.

[11] The Eclipse Foundation. *Graphical Modeling Framework (GMF)*, 2007. http://www.eclipse.org/gmf.

[12] M. D. D. Fabro, J. Bézivin, and P. Valduriez. Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany*, 2006.

[13] France Telecom. *SmartQVT: An open source model transformation tool implementing the MOF 2.0 QVT-Operational language*, 2007. http://smartqvt.elibel.tm.fr.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[15] A. Geburzi. Synthese von Modelltransformationsregeln aus Übersetzungsbeispielen. Master's thesis, Department of Computer Science, University of Paderborn, Nov. 2006.

[16] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards Verified Model Transformations. In D. Hearnden, J. G. Süß, B. Baudry, and N. Rapin, editors, *Proc. of the* 3^{rd} *International Workshop on Model Development, Validation and Verification (MoDeV*^{2}*a), Genova, Italy*, pages 78–93. Le Commissariat à l'Energie Atomique - CEA, Oct. 2006.

[17] H. Giese, M. Meyer, and R. Wagner. A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. In H. Giese and B. Westfechtel, editors, *Proc. of the* 4^{th} *International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*. University of Paderborn, Sept. 2006.

[18] J. Greenyer. A study of model transformation technologies: Reconciling TGGs with QVT. Master's thesis, Department of Computer Science, University of Paderborn, July 2006.

[19] J. Greenyer and E. Kindler. Reconciling TGGs with QVT, 2007. In preparation.

[20] C. Griffin. *Eclipse Model Transformation Framework (MTF)*. IBM, 2006. http://www.alphaworks.ibm.com/tech/mtf.

[21] E. Guerra and J. de Lara. Event-driven grammars: Towards the integration of meta-modelling and graph transformation. In *International Conference on Graph Transformation (ICGT'2004)*, volume 3265 of *LNCS*, pages 54–69, 2004.

[22] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *LNCS*, pages 321–335. Springer, Oct. 2006.

[23] A. Henriksson and H. Larsson. A definition of round-trip engineering. Technical report, Linkopings Universitet, Sweden, 2003.

[24] C. Hwan, P. Kim, and K. Czarnecki. Synchronizing cardinality-based feature models and their specializations. In A. Hartman and D. Kreische, editors, *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings*, volume 3748 of *Lecture Notes in Computer Science*, pages 331–348. Springer, Nov. 2005.

[25] IBM. *Rational Rose Developer for Java*, Mar. 2007. http://www-306.ibm.com/software/awdtools/developer/rose/java.

[26] I. Ivkovic and K. Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[27] E. Kindler, V. Rubin, and R. Wagner. An adaptable TGG interpreter for in-memory model transformation. In *Proc. of the Fujaba Days 2004*, pages 35–38, Darmstadt, Germany, Sept. 2004.

[28] E. Kindler, V. Rubin, and R. Wagner. Component Tools: Integrating Petri nets with other formal methods. In S. Donatelli and P. S. Thiagarajan, editors, *Application and Theory of Petri Nets 2006, 27^{th} International Conference*, volume 4024 of *LNCS*, pages 37–56. Springer, June 2006.

[29] A. Königs and A. Schürr. Tool integration with triple graph grammars - a survey. *Electronic Notes in Theoretical Computer Science*, 148(1):113–150, Feb. 2006.

[30] M. Lefering and A. Schürr. Specification of integration tools. In M. Nagl, editor, *Building Thightly-Integrated (Software) Development Environments: The IPSEN Approach*, volume 1170 of *LNCS*, pages 324–334. Springer Verlag, 1996.

[31] J. Leitner. Verifikation von Modelltransformationen basierend auf Triple Graph Grammatiken. Master's thesis, University of Karlsruhe, 2006.

[32] F. Marschall and P. Braun. Model transformations for the MDA with BOTL. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, CTIT Technical Report TR-CTIT-03-27, Univeristy of Twente, June 2003.

[33] OMG. *Model Driven Architecture*, 2003. http://www.omg.org/mda.

[34] OMG. *OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP*, 2003. http://www.omg.org/mda.

[35] OMG. *MOF QVT Final Adopted Specification, OMG Document ptc/05-11-01*, 2005. http://www.omg.org.

[36] OMG. *Unified Modeling Language: Superstructure Version 2.0*. Object Management Group, 140 Kendrick Street, Needham, MA 02494, USA, Aug. 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[37] Omondo. *EclipseUML Free Edition*, Feb. 2007. http://www.omondo.com.

[38] E. V. Paesschen, W. D. Meuter, and M. D'Hondt. SelfSync: A dynamic round-trip engineering environment. In L. C. Briand and C. Williams, editors, *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *LNCS*, pages 633–647. Springer, 2005.

[39] T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Model integration through mega operations. *Workshop on Model-driven Web Engineering*, 2005.

[40] A. Schürr. Specification of graph translators with triple graph grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, June 1994.

[41] S. Sendall and J. Küster. Taming model round-trip engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development, Vancouver, Canada*, Oct. 2004.

[42] G. Song, K. Zhang, and J. Kong. Model management through graph transformation. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 75–82, Washington, DC, USA, 2004. IEEE Computer Society.

[43] University of Paderborn, Germany. *Fujaba Tool Suite*, 2007. http://www.fujaba.de.

[44] D. Varró. Model transformation by example. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. 9th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2006, Genova, Italy, October 1-6, 2006.*, volume 4199 of *LNCS*, pages 410–424. Springer, Oct. 2006.

[45] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.

[46] G. Varró and D. Varró. Graph transformation with incremental updates. In R. Heckel, editor, *Proc. of the 4th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2004)*, volume 109 of *ENTCS*, pages 71–83. Elsevier, 2004.

[47] A. Vizhanyo, A. Agrawal, and F. Shi. Towards generation of efficient transformations. In G. Karsai and E. Visser, editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, volume 3286 of *Lecture Notes in Computer Science*, pages 298–316. Springer, 2004.

[48] W3C. XSL Transformations (XSLT) Version 1.0, November 1999. http://www.w3.org/tr/xslt.

[49] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards model transformation generation by-example. In *Proc. of the 40th Hawaii International Conference on System Sciences (HICSS'07), Hawaii, USA*, volume 0 of *System Sciences, 2007. HICSS 2007.*, page 285, Los Alamitos, CA, USA, Jan. 2007. IEEE Computer Society.