Lost in Translation? Transformation Nets to the Rescue!

Manuel Wimmer[1], Angelika Kusel[2], Thomas Reiter[2], Werner Retschitzegger[3], Wieland Schwinger[2], and Gerti Kappel[1]

[1] Vienna University of Technology, Austria
lastname@big.tuwien.ac.at
[2] Johannes Kepler University, Austria
lastname@ifs.uni-linz.ac.at
[3] University of Vienna, Austria
werner.retschitzegger@univie.ac.at

Abstract. The vision of Model-Driven Engineering places models as first-class artifacts throughout the software lifecycle. An essential prerequisite is the availability of proper transformation languages allowing not only code generation but also augmentation, migration or translation of models themselves. Current approaches, however, lack convenient facilities for debugging and ensuring the understanding of the transformation process. To tackle these problems, we propose a novel formalism for the development of model transformations which is based on Colored Petri Nets. This allows first, for an explicit, process-oriented execution model of a transformation, thereby overcoming the impedance mismatch between the specification and execution of model transformations, being the prerequisite for convenient debugging. Second, by providing a homogenous representation of all artifacts involved in a transformation, including metamodels, models and the actual transformation logic itself, understandability of model transformations is enhanced.

Key words: model transformation, runtime model, Colored Petri Nets

1 Introduction

Model-Driven Engineering (MDE) places models as first-class artifacts throughout the software lifecycle [3]. The main promise of MDE is to raise the level of abstraction from technology and platform-specific concepts to platform-independent and computation-independent modelling. To fulfil this promise, the availability of appropriate *model transformation languages* is *the* crucial factor, since transformation languages are for MDE as important as compilers are for high-level programming languages. Although numerous model transformation languages have been already proposed (for a survey, cf., [4]), currently no transformation language, not even the QVT (Query/View/Transformation) standard of the OMG [1], became accepted as the state-of-the-art model transformation language, i.e., an adoption in practice has not yet been achieved [7]. In particular, understandability and debuggability of model transformations are scarcely

supported by current approaches due to the following deficiencies. First, the artifacts involved in a model transformation, i.e., models, metamodels, as well as the actual transformation logic, are not represented in an *integrated view*. Instead, existing approaches only introduce formalisms for representing the transformation logic without considering the explicit representation of models and metamodels. Second, existing model transformation languages exhibit an inherent *impedance mismatch* between the specification and the execution of model transformations in terms of a one-to-many derivation of concepts. This is above all due to the fact that they do not support an explicit runtime model for the execution of model transformations which may be used to observe the runtime behavior of certain transformations [7], but rather execute their transformations on a low level of abstraction, e.g. based on a stack machine.

We therefore propose a novel formalism for developing model transformations called Transformation Nets [14], which tackles the aforementioned limitations of existing approaches. This formalism is based on Colored Petri Nets [8] and follows a process-oriented view towards model transformations. The main characteristic of the Transformation Net formalism is its ability to combine all the artifacts involved, i.e., metamodels, models, as well as the actual transformation logic, into one single representation. The possibility to gain an explicit, integrated representation of the semantics of a model transformation makes the formalism especially suited for gaining an understanding of the intricacies of a specific model transformation. This goes as far as to running the model transformation itself, as the Transformation Net constitutes a dedicated runtime model, thus serving as an execution engine for the transformation. This insight into transformation execution particularly favors the debugging and understanding of model transformations. Furthermore, Transformation Nets allow to build reusable modules that bridge certain kinds of structural heterogeneities, which are well-known in the area of database systems, as we have already shown in [10].

The remainder of this paper is structured as follows. Section 2 introduces the Transformation Net formalism. The subsequent Sections 3 and 4 present how the Transformation Net formalism is meant to be employed for concrete transformation problems by applying the formalism to a concrete example. Section 5 critically reflects the formalism by reporting on lessons learned from two case studies which have been conducted. Related work is discussed in Section 6. Finally, Section 7 gives a conclusion as well as an outlook on future work.

2 Transformation Nets at a Glance

This section introduces the Transformation Net formalism, whereby the conceptual architecture is shown in Figure 1. In this figure, we see a source metamodel on the left hand-side and a target metamodel on the right-hand side. In between, the transformation logic resides describing the correspondences between the metamodel elements. Furthermore, we see an input model conforming to the source metamodel, as well as an output model conforming to the target metamodel that represents the output of the transformation. The middle of Figure 1

shows the Transformation Net which represents the static parts of the transformation (i.e. metamodels and models) as places and tokens, respectively and the dynamic parts (i.e. the transformation logic) as appropriate transitions.

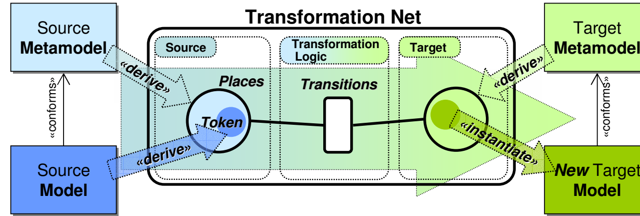

Fig. 1. Conceptual Architecture of Transformation Nets

Transformation Net Metamodel. The abstract syntax of the Transformation Net language is formalized by means of a metamodel (cf. Figure 2) conforming to the Ecore meta-metamodel, the Eclipse realization of OMG's MOF standard. This Transformation Net metamodel is based on Colored Petri Net concepts [8], but represents a specialized version thereof which aims at fulfilling the special requirements occurring in the domain of model transformations. In particular, in order to be able to encode metamodels and models, we introduce two kinds of places and two kinds of tokens (cf. Section 3). The second major adaption concerns the transitions. Since transitions are used to realize the actual transformation logic, we borrow a well established specification technique from graph transformation formalisms [6], which describe their transformation logic as a set of graphically encoded productions rules (cf. Section 4).

The whole Transformation Net metamodel is divided into four subpackages as can be seen in Figure 2. Thereby the package `Containers` comprise the modularization concepts. The package `StaticElements` is used to represent the static parts of a model transformation, i.e., metamodels and models. The dynamic elements, i.e., the actual transformation logic, are represented by concepts of the package `DynamicElements`. The package `Connectors` finally is responsible for connecting the static parts with the dynamic parts.

3 The Static Part of Transformation Nets

When employing Transformation Nets, in a first step, the static parts of a model transformation, i.e., the metamodels and models, need to be represented in our formalism (cf. package `StaticElements` in Figure 2). This incurs transitioning from the graph-based paradigm underlying MDE into the set-based paradigm underlying Petri Nets. The design rational behind this transition is the following: We rely on the core concepts of an object-oriented meta-metamodel, i.e., the graph which represents the metamodel consists of classes, attributes, and references, and the graph which represents a conforming model consists of objects,

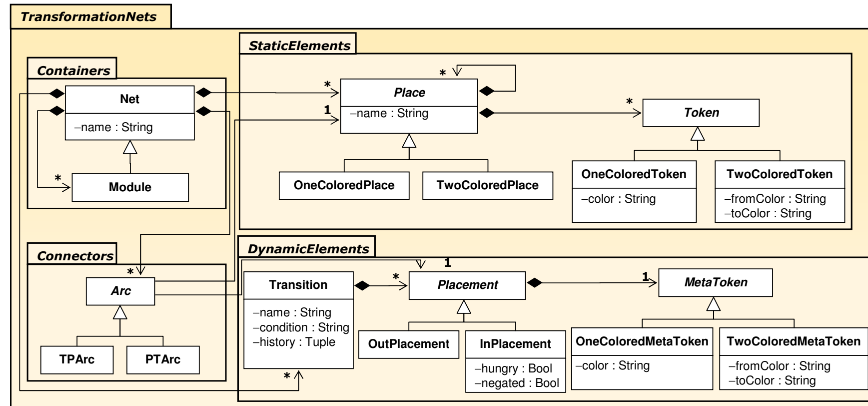

Fig. 2. The Transformation Net metamodel

data values and links. Therefore we distinguish between one-colored places containing one-colored tokens for representing the nodes of graphs, i.e., the objects, and two-colored places containing two-colored tokens. These two-colored tokens are needed for representing on the one hand links between the objects, i.e., one color represents the source object and the other the target object, and on the other hand attribute values, i.e., one color represents the containing object and the other the actual value.

Running Example. For describing the Transformation Net formalism in detail, we make use of a running example. The example is based on the Class2Relational case study[1] which became the de-facto standard example for model transformations, transforming an object-oriented model into a corresponding relational model. Due to reasons of brevity, only the most challenging part of this case study is described in this paper, namely how to represent inheritance hierarchies of classes within relational schemas.

Figure 3 shows UML diagrams of a simplified object-oriented metamodel as the source, and a metamodel for relational schemas as the target, together with a conforming source model and a to be generated target model. Thereby a 'one-table-per-hierarchy' approach is followed. Arguably, in terms of O/R mapping strategies, this may not be the most sophisticated approach. However, it makes the model transformation much more intriguing, thanks to the transitive closure that has to be computed over the class hierarchy. The middle layer of Figure 3 shows how the metamodel elements and model elements are represented as places and tokens, respectively, which is discussed in the following subsections.

3.1 Representing Metamodel Elements as Places

Classes represented as One-Colored Places. Both, abstract and concrete classes are represented as `OneColoredPlaces`. Subclass relationships are repre-

[1] http://sosym.dcs.kcl.ac.uk/events/mtip05

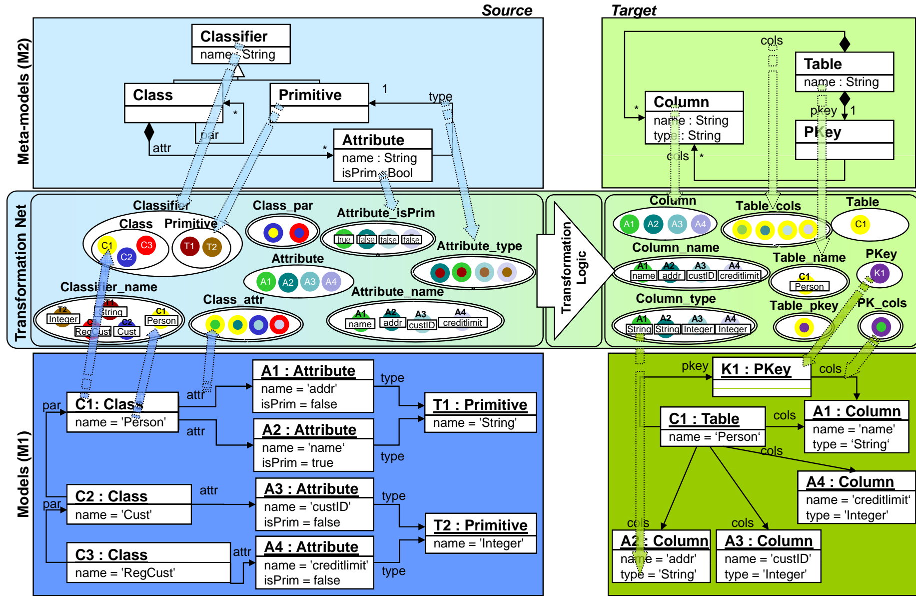

Fig. 3. The Class2Relational Transformation Problem

sented by the notion of places being contained within places. The notation used to visually represent one-colored places is a circle or an oval, which is traditionally used to depict places in Petri Nets. Concerning the example, depicted in Figure 3, one can see that each class of the metamodels – `Classifier`, `Primitive`, `Class` and `Attribute` of the source metamodel as well as `Table`, `Column` and `PKey` of the target metamodel – got represented through a respective one-colored place. Since `Class` and `Primitive` are subclasses of `Classifier`, these places became nested into the super-place.

References and Attributes represented as Two-Colored Places. References and attributes are represented as `TwoColoredPlaces`. Notationally, two-colored places are represented like one-colored places. However, for easier distinction, indicating that these places contain two-colored tokens, the borders of two-colored places are doubly-lined. Considering the running example, one can see that for each reference like e.g. `Attribute.type` and for each attribute like e.g. `Attribute.name` a corresponding two-colored place – `Attribute_type` and `Attribute_name`, respectively – has been created.

3.2 Representing Model Elements as Tokens

Objects represented as One-Colored Tokens. For every object, that occurs in a source model, a `OneColoredToken` is produced, which is put into the place that corresponds to the respective class in the source metamodel. The "color" is in fact expressed by means of a unique value that is derived from the identifying

attribute of the original model object. Hence, all one-colored tokens are "colored" with a unique literal. With regard to the running example, one can see that each instance of a class got represented through a respective one-colored token. Therefore, e.g. the one-colored place `Class` contains three one-colored tokens with distinct colors each one representing one of the three class instances `Person`, `Cust` and `RegCust`.

Links and Values represented as Two-Colored Tokens. For every link as an instance of a reference, as well as for every value as an instance of an attribute, a `TwoColoredToken` is produced. The `fromColor` attribute of such a token (cf. Figure 2) refers to the color of the token that corresponds to the owning object. The `toColor` is given by the color of the token that corresponds to the linked target object or the primitive data value. Notationally, a two-colored token is represented as a ring (denoting the "from"-color) surrounding an inner circle (denoting the "to"-color). Concerning the example, one can see that for each link as well as for each value a two-colored token got generated. Therefore, e.g. the two-colored place `Class_par` contains two tokens, in which one of these represents the inheritance relationship between the class `Cust` and the class `Person` and the other one represents the inheritance relationship between the class `RegCust` and the class `Cust`.

4 The Dynamic Part of Transformation Nets

After the previous section dealt with describing how models and metamodels are represented as the static parts of a Transformation Net, this section introduces the dynamic parts of a Transformation Net. The actual transformation logic is embodied through a system of Petri Net transitions and additional places which reside in-between those places representing the original input and output metamodels as is shown for the Class2Relational example in Figure 4. In this way, tokens are streamed from the source places through the Transformation Net and finally end up in target places. Hence, when a Transformation Net has been generated in its initial state, a specialized Petri Net engine can then execute the process and stream tokens from source to target places. The resulting tokens in the places that were derived from elements of the target metamodel are then used to instantiate an output model that conforms to the target metamodel.

Matching and producing model elements by firing transitions. An execution of a model transformation has two major phases. The first phase comprises the matching of certain elements of the source model from which information is derived that is used in the second phase for producing the elements of the output model. This matching and producing of model elements is supported within Transformation Nets by firing transitions. In Colored Petri Nets, the firing of a transition is based on a condition that involves the values of tokens in input places. Analogously, transitions in a Transformation Net are enabled if a certain configuration of matching tokens is available. This configuration is expressed

with the remaining elements of the previously shown Transformation Net metamodel (cf. subpackage *DynamicElements* in Figure 2). Thereby, transitions are represented through the `Transition` class. To specify their firing behavior, a mechanism well known from graph transformation systems [6] is used. Thereby, two patterns of input and output placeholders for tokens are defined, which represent a pre- and a post-condition by matching a certain configuration of tokens from the input places, and producing a certain configuration of tokens in the output places. The matching of tokens is the activity of finding a configuration of tokens from input places which satisfies the transition's pre-condition. Once such a configuration is found, the transition is enabled and ready to fire, with the colors of the input tokens to be bound to the input pattern. The production of output tokens once a transition fires, is typically dependent on the matched input tokens. For instance, when a transition is simply streaming a certain token, it would produce as an output the exact same token that was matched as an input token, thereby for example (cf. (c) in Figure 4) transforming an attribute of a top class into a column of the corresponding table.

Specification of transition's firing rules. In general, transformation rules are more complex than just transforming one element of the source model into exactly one element of the target model. Instead, to resolve structural heterogeneities only specific elements fulfilling certain conditions have to be selected, computations for deriving new values and links have to be done, and completely new elements have to be generated in the target model which do not exist in the source model. Considering our running example, such a complex transformation rule is e.g. that only top classes should be transformed into tables. For describing complex transition firing rules, we have chosen the following specification mechanism (cf. Figure 2). A transition can have a number of `Placement` objects. Such a placement is merely a proxy for a certain input or output place which is connected to the placement by an `Arc` object. The incoming and outgoing arcs of a transition are represented by the classes `PTArc` and `TPArc`, which connect to its owned `InPlacement` and `OutPlacement` objects. Every placement can then contain a `MetaToken` object, represented in the metamodel through the class `MetaToken` and its specializations `OneColoredMetaToken` and `TwoColoredMetaToken`. Hence, a meta token can either stand for a one-colored or a two-colored token and can be used in two different ways:

- **Query Tokens**: Query tokens are meta tokens which are assigned to input placements. Query tokens can either stand for one-colored or two-colored token configurations, whose colors represent variables that during matching are bound to the color of an actual input token. Note that the colors of query tokens are not the required colors for input tokens, instead they describe color combination patterns that have to be fulfilled by input tokens. Normally, query tokens match for existence of input tokens but with the concept of negated input placements it is also possible to check for the non-existence of certain tokens. For example, this is required in our running example to find top classes, because a top class is a class which does not have an outgoing `par` link to another class.

- **Production Tokens**: Output placements contain so-called production tokens which are equally represented through the class `MetaToken` and its subclasses. For every production token in an output placement, a token is produced in the place that is connected to the output placement via an outgoing arc. The color of the produced token is defined by colors that are bound to the colors of the input query tokens. However, it is also possible to produce a token of a not yet existing color, for instance if the color of the output query token does not fit to any of the input query tokens. With this mechanism, new elements can be created in the target model which do not exist in the source model. Considering our running example, this mechanism is needed in order to produce primary keys in the target model which are not explicitly represented in the source model.

Please note that the default firing behavior of Transformation Nets is different to standard Petri Nets in the sense that transitions in standard Petri Nets always consume the tokens from the input places and produce new tokens in the output places. This behavior can also be achieved in Transformation Nets by setting the value of the attribute `hungry` of the corresponding `Inplacement` to "true". It has to be emphasized, however, that this is not the default setting due to the fact that it is often the case that more than one transition has a certain place as an input place and therefore if all the connected transitions would consume the tokens, erroneous race conditions would appear. Therefore, by default, every transition is just reading the tokens of the connected input places and does not delete them. In order to prevent a transition to fire more than once for a certain token configuration, the already processed configurations are stored in a so-called switching history (cf. attribute `history` in Figure 2). In our running example, all transitions are marked as being not hungry.

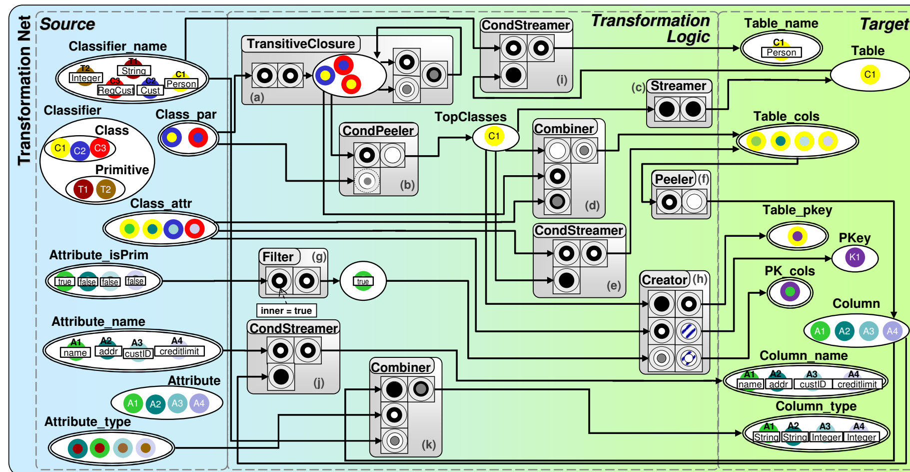

Fig. 4. Class2Relational example realized with Transformation Nets

Transformation Logic for the Class2Relational Example. To exemplify the use of transitions for defining transformation logic, Figure 4 depicts the transitions necessary in our running example. Thereby the transformation net is shown in it's final state after streaming all the tokens through the net. As mentioned in Section 3, a 'one-table-per-hierarchy' approach is pursued resulting in the need for computing the transitive closure, i.e. making explicit all inheritance relationships, of the class hierarchy. Module (a) contains two transitions and a place which are jointly responsible for fulfilling this task. Thereby the left transition just copies the inheritance links defined in the model by matching a two-colored token from the input place `Class_par` and streaming this token to the connected output place. This output place accumulates all inheritance links including the transitive ones that are computed by the following right transition. This transition takes two two-colored tokens, each one representing a parent-link of the inheritance hierarchy and if the inner color of the one input-token matches the outer color of the other input token, i.e., there is another class that inherits from the subclass, a link is established from this indirect subclass to the superclass and put into the corresponding place. In this way, all possible inheritance relationships can be derived. The ones that have no further parent are extracted and matched by the transition of module (b). Note that for this transition we have to use a negated input placement represented by a dashed circle. If such a matching token configuration is found, the transition takes the inner color of this link and streams it to the `TopClasses` place, since such a token represents a top-level-class of the inheritance hierarchy. These top-level-classes are of special interest in this transformation as the number of connected transitions to this place reveals. Module (c), for instance, is responsible for creating tables for each found top-level class and therefore just streams tokens of this place to the `Table` place. Modules (d) and (e) are responsible for computing the columns of the generated tables and therefore also rely on the top-level classes. In order to accomplish this task, module (e) streams all those `Class_attr` tokens to the `Table_cols` place that are owned by a top-level class. Additionally since a 'one-table-per-hierarchy' approach is followed, also those attributes need to become columns which are contained in some subclass and for this task module (d) is responsible. Thereby all those `Class_attr` tokens are streamed to the `Table_cols` place which are in a direct or indirect inheritance relationship to a top-level-class according to the transitive closure. From these two-colored tokens, module (f) generates tokens for the `Column` place by peeling the inner color out of the two-colored tokens and generating one-colored output tokens. For generating primary keys from identifier attributes, i.e., attributes where `isPrim = true`, module (g) and (h) are employed. While the first one is a special kind of conditional streamer for filtering the identifier attributes by evaluating the condition (`inner = true`) and assigning the result token to the transition instead of an additional query token, the second one is responsible for generating for each identifier attribute a new one-colored token representing a primary key and linking this newly created token with tables and columns accordingly. Finally, module (i), (j), and (k)

are used to stream the attribute values for the previously generated tables and columns into the places `Table_name`, `Column_name`, and `Column_type`.

5 Lessons Learned

This section presents lessons learned from the Class2Relational case study. Additionally to this horizontal (i.e., model to model) transformation scenario, a vertical (i.e., model to code) transformation scenario, has been conducted in order to clarify the value of our approach for diverse application areas. The vertical scenario is the BPMN2BPEL example taken from a graph transformation tool contest[2]. The case studies have been realized with the help of our prototype for modeling, executing and debugging transformation nets. Further details of the case study realizations and tool support may be found at our project page[3].

Composition and weak typing favors reusability. First of all, it has been shown that several kinds of transitions occur many times with minor deviations only. Such transitions can be generalized to transformation patterns and subsequently realized by modules. Since the inplacements as well as the outplacements are just typed to one-colored tokens and two-colored tokens, respectively and not to certain metaclasses, these modules can be reused in different scenarios. This kind of reuse is not restricted to single transitions only, since through the composition of transitions by sequencing as well as nesting the resulting modules, modules, realizing complex transformation logic, can be built. The Class2Relational case study was realized by the usage of just eight different modules, whereby the `CondStreamer` module was applied three times (cf. Figure 4), thus justifying the potential for reuse.

Visual syntax and live programming fosters debugging. Transformation nets represent a visual formalism for defining model transformations which is especially favorable for debugging purposes. This is not least since the flow of model elements undergoing certain transformations can be directly followed by observing the flow of tokens whereby undesired results can be detected easily. Another characteristic of transformation nets, that fosters debuggability, is live programming, i.e., some piece of transformation logic can be executed and thus tested immediately after definition without any further compilation step. Therefore, testing can be done independently of other parts of the Transformation Net by just setting up a suitable token configuration in the input places.

Implicit control flow eases evolution. The control flow in a transformation net is given through data dependencies between various transitions. As a consequence, when changing a transformation, one needs to maintain a single artifact only instead of requiring additional efforts to keep control flow and transformation logic (in the form of rules) synchronized. For instance, when a certain rule would need to be changed to match for additional model objects,

[2] http://www.fots.ua.ac.be/events/grabats2008

[3] http://big.tuwien.ac.at/projects/tropic

one has to explicitly take care to call this rule at a time when the objects to be matched already exist.

Fine-grained model decomposition facilitates resolution of heterogeneities. The chosen representation of models by Petri nets lets references as well as attributes become first-class citizens, resulting in a fine-grained decomposition of models. The resulting representation in combination with weak typing turned out to be especially favorable for the resolution of structural heterogeneities. This is since on the one hand there are no restrictions, like a class must be instantiated before an owned attribute and since on the other hand e.g. an attribute in the source model can easily become a class in the target model by just moving the token to the respective place.

Transitions by color-patterns ease development but lower readability. Currently the precondition as well as the postcondition of a transition are just encoded by one-colored as well as two-colored tokens. On the one hand, this mechanism eases development since e.g. for changing the direction of a link it suffices just to swap the respective colors of the query token and the production token. On the other hand, the case study has shown that the larger the transformation net grows the less readable this kind of encoding gets. Therefore, it has been proven useful to assign each input as well as each output placement a human-readable label, that describes the kind of input and output, respectively.

6 Related Work

One of the main objectives of transformation nets is to enhance the debuggability and understandability of model transformations by using a Petri net based formalism. Therefore, we consider two orthogonal threads of related work. First, we discuss how debugging and understandability in terms of a visual representation as well as the possibility for graphical simulation are supported by current model transformation approaches, and second, we elaborate on the general usage of Petri Nets in model transformation approaches.

Debugging Support and Understandability. Debugging support only at the execution level requires traceability to the corresponding higher-level mapping specifications in order to be aware of the effects a certain mapping induces on the state of the execution. For example, in the Fujaba environment[4], a plugin called MoTE [16] compiles TGG rules [11] into Fujaba story diagrams that are implemented in Java, which obstructs a direct debugging on the level of TGG rules. Additional to that, Fujaba supports visualization of how the graph evolves during transformation, and allows interactive application of transformation rules. Furthermore, approaches like VIATRA [2] producing debug reports that trace an execution, only, are likewise considered inadequate for debugging since a minimum requirement for the debugging should be the ability to debug at least whole transformation rules, by which we refer to as the stepwise execution and inspection of the execution state. The debugging of ATL [9] is based on

[4] http://www.fujaba.de

the step-wise execution of a stack-machine that interprets ATL byte-code, which also allows observing the execution of whole transformation rules. SmartQVT[5] [1], TefKat [12] and KerMeta [13] allow for similar debugging functionality.

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

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

Petri Nets and Model Transformations. The relatedness of Petri nets and graph rewriting systems has also induced some impact in the field of model transformation. Especially in the area of graph transformations some work has been conducted that uses Petri nets to check formal properties of graph production rules. Thereby, the approach proposed in [15] translates individual graph rules into a place/transition net and checks for its termination. Another approach is described in [5], which applies a transition system for modeling the dynamic behavior of a metamodel.

Compared to these two approaches, our intention to use Petri nets is entirely different. While these two approaches are using Petri nets as a back-end for automatically analyzing properties of transformations, we are using Petri nets as a front-end for fostering debuggability and understandability. In particular, we are focussing on how to represent model transformations as Petri Nets in an intuitive manner. This also covers the compact representation of Petri Nets to eliminate the scalability problem of low-level Petri nets. Finally, we introduce a specific syntax for Petri Nets used for model transformations and integrate several high-level constructs, e.g., inhibitor arcs and pages, into our language.

7 Conclusion and Future Work

In this paper, we have presented the Transformation Net formalism which is meant to be a runtime model for the representation of model transformations. First investigations have shown that the formalism is promising to solve a wide spectrum of transformation problems like horizontal transformation scenarios

[5] http://smartqvt.elibel.tm.fr

and vertical transformation scenarios, respectively. Especially the debugging of model transformations is fostered since Transformation Nets provide an integrated view on all transformation artifacts involved as well as a dedicated runtime model. For future work we strive to investigate formal properties like reachability, liveness or boundedness of Petri Nets and their potential applicability as well as usefulness for model transformations. Furthermore we aim at translating existing model transformation languages into transformation nets like the QVT-Relations standard. By doing so, we gain (1) operational semantics for the QVT-Relations standard and (2) a visual debugging possibility.

References

1. Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification, 2007.
2. A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. *21st ACM Symposium on Applied Computing*, 2006.
3. J. Bézivin. On the Unification Power of Models. *Journal on Software and Systems Modeling*, 4(2), 2005.
4. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 2006.
5. J. de Lara and H. Vangheluwe. Translating Model Simulators to Analysis Models. *11th Int. Conf. on Fundamental Approaches to Software Engineering*, 2008.
6. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools.* World Scientific Publishing Co., 1999.
7. R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *29th Int. Conf. on Software Engineering*, 2007.
8. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Monographs in Theoretical Computer Science, Springer, 1992.
9. F. Jouault and I. Kurtev. Transforming Models with ATL. *Model Transformations in Practice Workshop of MODELS'05*, 2005.
10. G. Kappel, H. Kargl, T. Reiter, W. Retschitzegger, W. Schwinger, M. Strommer, and M. Wimmer. A framework for building mapping operators resolving structural heterogeneities. *2nd Int. United Information Systems Conf. (UNISCON'08)*, 2008.
11. A. Koenigs. Model Transformation with Triple Graph Grammars. *Model Transformations in Practice Workshop of MODELS'05*, 2005.
12. M. Lawley and J. Steel. Practical Declarative Model Transformation with Tefkat. *Model Transformations in Practice Workshop of MODELS'05*, 2005.
13. P. Muller, F. Fleurey, and J. Jezequel. Weaving Executability into Object-Oriented Meta-languages. *8th Int. Conf. on Model Driven Engineering Lanuages and Systems*, 2005.
14. T. Reiter, M. Wimmer, and H. Kargl. Towards a runtime model based on colored Petri-nets for the execution of model transformations. In *Proceedings of the 3rd Workshop on Models and Aspects @ ECOOP07*, 2007.
15. D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. *3rd Int. Conf. on Graph Transformations*, 2006.
16. R. Wagner. Developing Model Transformations with Fujaba. *4th Int. Fujaba Days*, 2006.