Lets's Play the Token Game – Model Transformations Powered By Transformation Nets*

M. Wimmer[1], A. Kusel[2], J. Schoenboeck[1], T. Reiter[2],
W. Retschitzegger[3], and W. Schwinger[2]

[1] Vienna University of Technology, Austria
{wimmer,schoenboeck}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
kusel@bioinf.jku.at, reiter@ifs.uni-linz.ac.at
wieland.schwinger@jku.ac.at
[3] University of Vienna, Austria
werner.retschitzegger@univie.ac.at

Abstract. Model-Driven Engineering (MDE) is a software engineering paradigm using abstract models to describe systems which are then systematically transformed to concrete implementations. Since model transformations are crucial for the success of MDE, several kinds of dedicated transformation languages have been proposed. Hybrid languages combine the statefulness and the ability to define control flow of imperative approaches with the raised level of abstraction of declarative ones. However the low-level engines employed to execute transformations lead to an impedance mismatch between specification and execution of model transformations which hampers debugging. Additionally, current approaches lack of appropriate reuse mechanisms for resolving recurring transformation problems. Therefore, we propose a process-oriented specification and execution of model transformations based on Transformation Nets, a variant of Colored Petri Nets (CPNs). By using Transformation Nets, the benefits of imperative and declarative approaches are combined, not only for the specification of model transformations, but also for their execution by using CPNs themselves as a transformation engine. Furthermore, Transformation Nets introduce a novel notation for implementing transformation logic within transitions to foster reuse.

Key words: Model-Driven Engineering, Model Transformations, CPN

1 Introduction

Model-Driven Engineering (MDE) [7] is a current trend in software engineering where models are used to describe systems which are then systematically transformed to concrete implementations. Thus, MDE places models as first-class

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

artifacts throughout the whole 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 fulfill 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. Transformation scenarios can be divided into vertical model transformations and horizontal model transformations. Vertical model transformations lower the level of abstraction, e.g., transforming UML class diagrams to relational models, whereas horizontal model transformations transform models between two different representations on the same level of abstraction, which is the focus of our approach, e.g., a UML class model is transformed to an entity relationship diagram.

To specify model transformations, approaches range from purely imperative styles allowing to define how a transformation is carried out to fully declarative transformation definition styles focusing on what a transformation's output should be like according to a certain input. In between this spectrum, hybrid approaches combine the statefulness and the ability to define control flow of imperative approaches with the raised level of abstraction of declarative ones [4]. In general, declarative and hybrid approaches use transformation engines to execute the model transformations operating on a low level of abstraction, e.g., the Atlas Transformation Language (ATL) uses a stack machine [9], shown as black-box to the transformation designer. As a consequence, debugging of model transformations is limited to the information provided by the transformation engine, only, most often consisting of variable values and logging messages, but missing important information, e.g., why a certain part of a transformation is actually executed or not. This is due to the fact that an explicit runtime model [7] for the execution of model transformations is not supported which could be used to observe the runtime behavior of certain transformations.

Another problem current transformation languages suffer from is that no appropriate reuse mechanisms and pre-defined libraries for resolving recurring model transformation problems are provided. Especially, the resolution of structural heterogeneities [10], i.e., the same concept is expressed by different metamodel elements, represents a recurring challenge. Thus, resolving structural heterogeneities requires to manually specify partly tricky model transformations again and again for each scenario.

The contribution of this paper is a novel approach for defining model transformations called Transformation Nets [12], which use a variant of Colored Petri Nets (CPNs) [8]. Transformation Nets embody a process-oriented view on model transformations, whereby the actual transformation is carried out by reuasable patterns of transformation logic that stream models represented by the net's tokens from source to target. Such a runtime model provides the explicit statefulness of imperative approaches through tokens contained in the net's places. The abstraction of control flow from declarative approaches is achieved as the net's transitions can fire autonomously depending on their environment and effectively make use of implicit, data-driven control flow. Furthermore, Trans-

formation Nets provide a uniform formalism not only for representing the transformation logic together with the metamodels and the models themselves, but also for executing the transformations. Thus, no impedance mismatch between specification and execution occurs which allows for enhanced understandability and debuggability of model transformations. In this paper we present how a variant of CPN is employed for the domain of model transformations. Firstly, we show how model transformations are appropriately represented with a variant of CPNs, and secondly, a novel notion for implementing reusable transformation logic within transitions is proposed.

The remainder of this paper is structured as follows. Section 2 gives an overview of the Transformation Net formalism. While, the subsequent Section 3 describes the static part of Transformation Nets, depicting how metamodels and models are represented within Transformations Nets, Section 4 elaborates on the dynamic part of Transformation Nets, especially how transformation logic is specified. Section 5 reports on lessons learned by the application of Transformation Nets. Related work is discussed in Section 6, and finally, Section 7 gives a conclusion and an outlook on future work.

2 Transformations Nets at a Glance

Within this section the basic concepts of model transformations in general and Transformation Nets in particular are introduced.

2.1 Big Picture of Model Transformations

The general model transformation scenario is illustrated in the upper half of Figure 1 comprising a transformation with one input (source) model and one output (target) model [4]. Both models conform to their respective metamodels which define the abstract syntax of a modeling language. Transformation Nets provide an integrated view on model transformations by homogenously representing metamodels, models, and transformation logic. At the same time Transformation Nets serve also as a runtime model to actually carry out the transformation. One can differentiate between static and dynamic parts of a Transformation Net. The static parts correspond to the transformations' metamodel elements (represented as places), given input model elements and generated output model elements (represented as tokens), whereas the dynamic parts corresponds to the transformation logic itself (represented as transitions) as formalized in a metamodel introduced in the following.

2.2 Transformation Net Metamodel

The abstract syntax of the Transformation Net is formalized by means of a metamodel (cf. Fig. 2) conforming to the OMG's Meta Object Facility (MOF) [1] standard. The Transformation Net metamodel describes appropriately adapted

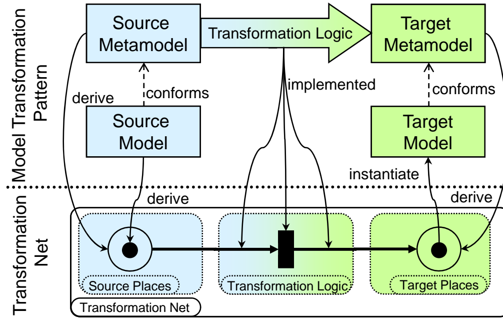

Fig. 1. Conceptual Architecture of Transformation Nets

Colored Petri Net concepts [8] in order to fulfill 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. Sec. 3). The second major adaption concerns the transitions. Since transitions are used to realize the 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. Sec. 4).

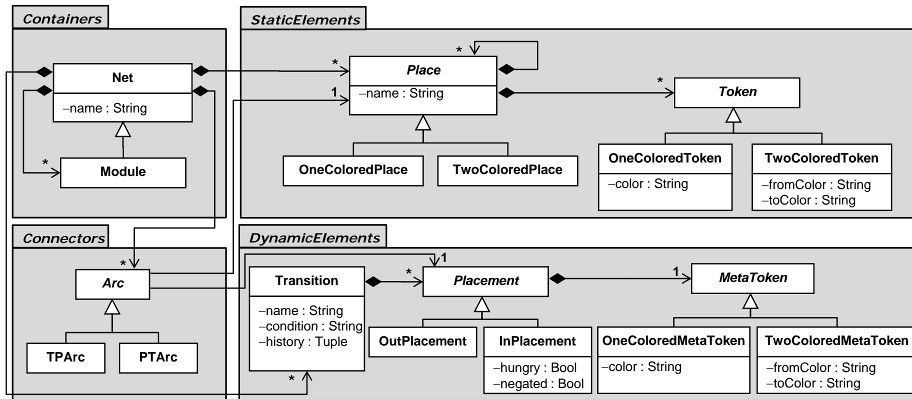

Fig. 2. The Transformation Net Metamodel

The whole Transformation Net metamodel is divided into four subpackages as can be seen in Fig. 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 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. In the following sections, we describe the packages and their contained elements in detail.

3 The Static Part of Model Transformations

To represent metamodels and models in Transformation Nets a translation from the graph-based paradigm underlying MDE to the set-based paradigm underlying Petri Nets is necessary. We rely on the core concepts of an object-oriented meta-metamodel as defined by MOF, being classes, attributes, and references. In the following, we elaborate on how MOF concepts used in metamodels and their respective instances are represented within Transformation Nets (cf. Fig. 3 for a summary).

	Meta Object Facility (MOF)		Transformation Nets		CPN
	Concept	Example	Concept	Example	Example
Metamodel Elements	Class	Array / id : String	OneColoredPlace	Array	Array / OneColoredPlace
	Attribute	Array / id : String	TwoColoredPlace	Array_id	Array_id / TwoColoredPlace
	Reference	Array / id : String —contains *→ Element / id : String	TwoColoredPlace	Array_contains	Array_contains / TwoColoredPlace
	Generalization	Collection ◁— Array / id : String	NestedPlace	Collection / Array	Not directly supported
	Ordered Reference	Array / id : String —{ordered} *→ Element / id : String	OrderedPlace	Array_contains / ordered	Not directly supported
	Not used in metamodels		Absolute Capacity	Buffer / 1	Not directly supported
	Relative Capacity	Array / id : String —head 1→ Element / id : String	Relative Capacity	Array_head / 1	Not directly supported
Model Elements	Object (Instance of Class)	P1:Array / id = \`Paper1´	OneColoredToken (contained in a OneColoredPlace)	Array / P1	1\`(1,"P1") / Array / OneColoredPlace
	Value (Instance of Attribute)	P1:Array / id = \`Paper1´	TwoColoredToken (contained in a TwoColoredPlace)	Array_id / P1 / Paper1	1\`(1,"P1","Paper1") / Array_id / TwoColoredPlace
	Link (Instance of Reference)	P1:Array / id : \`Paper1´ —contains→ S1:Element / id : \`Section1´	TwoColoredToken (contained in a TwoColoredPlace)	Array_contains / P1 / S1	1\`(1,"P1","S1") / Array_contains / TwoColoredPlace

Fig. 3. Representing MOF concepts within Transformation Nets

3.1 Representing Metamodel Elements as Places

A metamodel mainly consists of classes, attributes, and references which are mapped to places in a Transformation Net. We distinguish between two kinds of places, namely `OneColoredPlace` to represent classes, and `TwoColoredPlace` to represent attributes and references. Both types of places can be represented by according data types in CPNs which are assigned to the corresponding places.

Classes represented as One-Colored Places. Abstract and concrete classes are both represented as `OneColoredPlaces`. Although, abstract classes cannot have instances, places created from abstract classes normally contain tokens indirectly due to other places stemming from sub-classes, (cf. below) being contained within them. Furthermore, the name of the class becomes the name of the place (`name`). The notation used to visually represent one-colored places is

an oval as traditionally used to depict places in Petri nets. Subclass relationships are represented by `nestedPlaces` whereby the place corresponding to the subclass is contained within the place corresponding to the superclass. The tokens contained in the"sub-place" are also visible in the "super-place", which means that if a token is contained in a sub-place it may also act as input token for a transition connected to the "super-place".

Attributes and References represented as Two-Colored Places. Attributes and references are represented by `TwoColored Places`, whereby the name of the place consists of the name of the containing class and the name of the attribute or reference itself, separated by an underscore (`ClassName_name`). Notationally, the borders of two-colored places are doubly-lined to indicate that they contain two-colored tokens.

Orderings. References that impose an order on their links, e.g., an array element which has several elements contained in a specific order, require some extensions to normal two-colored places. If in case of such an ordered reference, the content of the two-colored place is internally set up as several ordered sequences (but not explicitly represented). For instance, for each different array represented by different `fromColor`, a sequence of two-colored tokens with that `fromColor` exists. Sequences in ordered places are working according to the FIFO principle in order to facilitate the implementation of transformation logic.

Multiplicities. Places can restrict the amount of tokens they can contain. In particular two-colored places have an absolute capacity (`absCapacity`) to restrict the total number of its tokens and a relative capacity (`relCapacity`) to restrict the maximum length of its sequences. Hence, multiplicities of references can be mapped onto the relative capacity of a two-colored place. For instance, a two-colored place with a capacity of '1' may contain several tokens, but for each token a distinct `fromColor` is mandatory. An absolute capacity would allow only one token irrespective of its color inside the place, e.g., used to ensure a sequential eradication or to represent singleton classes. Capacities are visualized through the respective number inside the place, which is underlined in case of an absolute capacity.

3.2 Representing Model Elements as Tokens

Objects represented as One-Colored Tokens. For every object, i.e., instance of `Class` that occurs in a model a `OneColoredToken` is produced, which is put into a place that corresponds to the respective element in the source metamodel. The "color" is realized through a unique value (`color`) that is derived from the object_id (OID).

Values and Links represented as Two-Colored Tokens. For every value as an instance of an `Attribute`, as well as for every link as an instance of a `Reference`, a `TwoColoredToken` is produced. The `fromColor` attribute refers to the color of the token (thus the OID) that corresponds to the owning object. The `toColor` is given by the color of the token that corresponds to the referenced target object (the OID of the target object). Notationally, a two-colored token

is represented as a ring (depicting the `fromColor`) surrounding an inner circle (depicting the `toColor`).

3.3 Transformation Nets by Example

By making use of an example we show how Transformation Nets can be applied to transform arrays to linked lists. The top of Fig. 4 depicts the source and target metamodels, as well as the input model and the desired, semantically equivalent output model. The mapping of concepts between the array metamodel and the linked list metamodel is achieved by transforming the `Array` class to an equivalent `LinkedList` class, just like the `Element` classes are transformed to `Node` classes. The ordered set of `contains` links needs to be translated into correctly set up `head`, `next` and `prev` links that maintain a semantically equivalent ordering of `Node` objects.

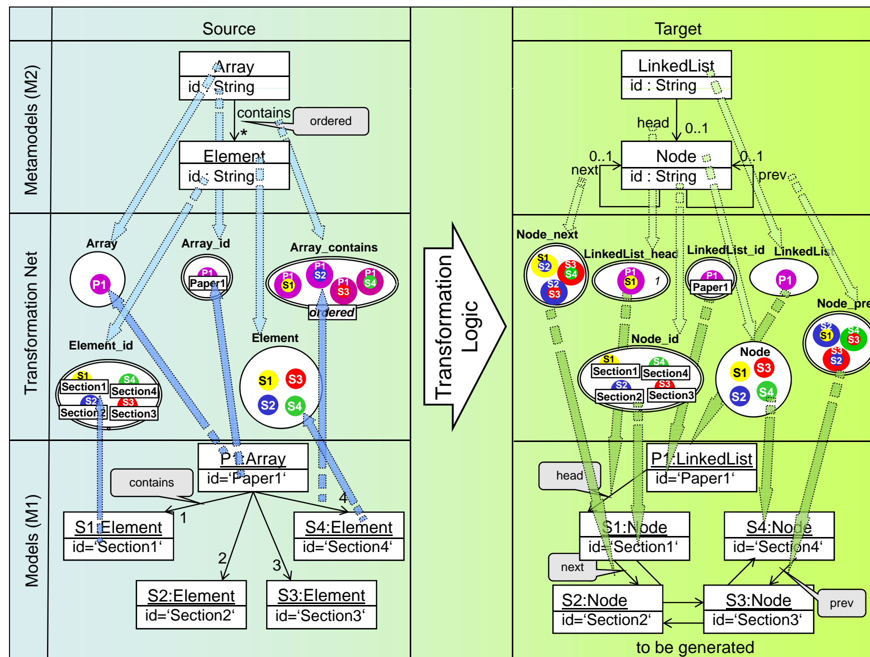

Fig. 4. Motivating Example: Static Part

Fig. 4 depicts the static parts of the Transformation Net for the example. The metamodel elements `Array` and `Element` are represented by two corresponding places in the Transformation Net. Both classes have an `id` property, represented as two-colored places as well as the reference `contains`. The lower part of the figure shows models conforming to the metamodels used to extract tokens which are put into the corresponding places. The array `P1` is indicated by a one-colored token in the `Array` place whereas the `id` of the array is depicted by a two-colored

token in the `Array_id` place. The four elements (`S1` to `S4`) contained in the `Array` are represented by four according tokens in the `Element` place. In addition four two-colored tokens are created representing the `ids` of the attributes and put into the `Element_id` place. The four tokens in the place `Array_contains` represent the links between array (depicted by the outer color) and it's elements (depicted by their inner color). Analogous to the source model, places for the target model are created which are empty until the target tokens are generated by executing the model transformation. The shown target tokens in Fig. 4 are therefore the result of a successfully executed transformation, which are then used to instantiate the target model.

4 The Dynamic Part of Model Transformations

The transformation logic is embodied by a system of Petri Net transitions and additional places which reside in-between those places representing the original input and output metamodels. In this way, tokens are streamed from the source places through the Transformation Net and finally end up in the target places. Hence, when a Transformation Net has been generated in its initial state, i.e., source tokens are already derived from the input model, a specialized Petri Net engine executes the transformation process and streams tokens from source to target places. The resulting tokens in the target places are then used to instantiate an output model that conforms to the target metamodel. In the following it is shown how to actually match for source model elements and how target model elements are produced.

4.1 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 whereas the second phase produces the elements of the output model. This matching and producing of model elements is supported within Transformation Nets by firing transitions. To specify their firing behavior, a mechanism well known from graph transformation systems is used [6]. Transitions consist of input placeholders (LHS of the transition) representing the pre-conditions of a certain transformation, whereas output placeholders (RHS of the transition) depict its post-condition. Those placeholders are expressed by the classes `InPlacement` (LHS) and `OutPlacement` (RHS) in the metamodel as shown in Fig. 2. Every `Placement` is connected to a source or target place using `Arcs`, whereby incoming and outgoing arcs are represented by the classes `PTArc` and `TPArc`, respectively. To express these pre- and post-conditions, so-called meta tokens (cf. class `MetaToken` in the metamodel) are used, prescribing a certain token configuration by means of color patterns which can be used in two different ways, either as Query Token (LHS) or as Production Token (RHS), as shown in Fig. 5.

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 are bound during matching 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 configurations that have to be fulfilled by input tokens. Normally, query tokens match for the 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 (cf. attribute `negated` of class `InPlacement` in Fig. 2).

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 contained in one transition. 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.

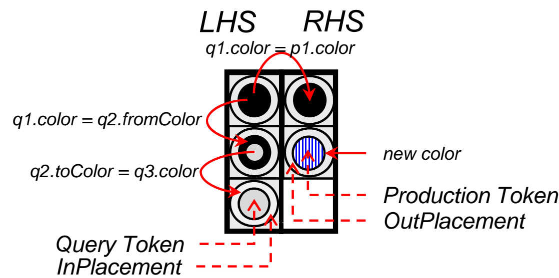

Fig. 5. Example Transition and Color Binding

By matching a certain token configuration from the input places, the transition is ready to fire with the colors of the input tokens bound to the meta tokens residing in the input placements. The production of output tokens once a transition fires is dependent on the matched input tokens. For example, when a transition is simply streaming a certain token, it would produce as an output the exact same token that was matched as the input token (cf. (a) in Figure 6). This form of transition is especially needed for implementing one-to-one correspondences between metamodel elements. Please note that the default firing behavior of a Transformaion Net does not consume the tokens of the input places in order to avoid race conditions as often several transitions make use of one and the same input place. This can be changed by setting the attribute `hungry` of the corresponding `InPlacement` to "true". In order to prevent a transition to fire more than once for a certain token configuration, the already processed configurations are stored in the `history` of a transition.

4.2 Reusable Transformation Logic

Since `InPlacements` as well as `OutPlacements` are just typed to one-colored tokens and two-colored tokens, but not to certain metamodel classes, these transitions can be reused in different scenarios. Different to CPNs which use arc-inscriptions to encode firing behavior as shown in Fig. 6, Transformation Net transitions encapsulate this information. In Transformation Net arcs need no inscriptions and therefore places extracted from a metamodel can be connected directly to predefined transitions. 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 transformation modules realize complex transformation logic. Furthermore, the graphical representation of pre- and post-conditions by color patterns is similar to graph transformation patterns transformation designers are used to. To exemplify basic firing rules, Fig. 6 shows a series of transitions with their placements containing patterns of one- and two-colored query and production tokens and their histories are shown below the actual transition as well as the equivalent transitions in CPNs. These transitions represent reusable patterns which are applied in the following to solve the example of Sec. 4.3.

	(a) Streamer	(b) Inverter	(c) Linker	(d) Peeler	(e) Conditional Streamer
Example	A, X, History	A, X, History	A, B, X, History	A, X, History	A, B, X, History
CPN	A –x→ □ –x→ X	A –(x,y)→ □ –(y,x)→ X	A –x→, B –y→ □ –(x,y)→ X	A –(x,y)→ □ –y→ X	A –x→, B –(x,y)→ □ –x→ X
Pre Cond	$\exists Aa \in A$ *Place A contains a one colored token **a***	$\exists Aa \in A$ *Place A contains a two colored token **a***	$\exists Aa \in A \wedge \exists Bb \in B$ *Place A and B contain a one colored token **a** and **b***	$\exists Aa \in A$ *Place A contains a two colored token **a***	$\exists Aa \in A \wedge \exists Bb \in B[col(a) = from(a)]$ *Place A and B contain a one colored token **a** and **b***
Post Cond	$\exists Xx \in X[col(x) = col(a)]$ *Place X contains a token **x** whose color is the same as the matched token **a**.*	$\exists Xx \in X[from(x) = to(a) \wedge to(x) = from(a)]$ *Place X contains a token **x** whose colors are inverted to those of the matched token **a**.*	$\exists Xx \in X[from(x) = col(a) \wedge to(x) = col(b)]$ *Place X contains a token **x** whose 'from' color is the color of the matched token **a** and whose 'to' color is the color of the matched token **b**.*	$\exists Xx \in X[col(x) = to(a)]$ *Place X contains a token **x** whose 'color is the same as the 'to' color of the matched token **a***	$\exists Xx \in X[col(x) = col(a)]$ *Place X contains a token **x** whose color is the same as the matched token **a**.*

Fig. 6. Reusable Transformation Logic expressed as Transitions

`Transition (a)` in Fig. 6 shows a simple pattern that matches a one-colored token from an input place and streams the exact same token to an output place, therefore this pattern is called `Streamer`. It can be applied in cases of one-to-one mappings, e.g., an array is transformed to a linked list. `Transition (b)` matches a two-colored token from its input place, and produces an inverted token in the output place. This `Inverter` pattern can be applied to set inverse

references (cf. "next" and "prev" references in our example). `Transition (c)` called `Linker` shows two one-colored query tokens on the input side, whose matched colors determine the "from"- and "to"-colors of the produced output token. This pattern is used to introduce new links between objects. `Transition (d)` matches two-colored tokens from input places and peels off the outer color of the token, therefore the name `Peeler`. This pattern is used to get the value of an attribute or the target of a link which is represented by the inner color of the two-colored token. Finally, `transition (e)` represents a variation of the `Streamer` pattern called `ConditionalStreamer` adding additional query tokens to ensure certain preconditions. For example, this pattern may be used to ensure that before a link between two objects is set the objects to be linked have been created.

4.3 Solution for the Motivating Example

Fig. 7 depicts the complete Transformation Net realizing the necessary transformation logic added in between the static parts of source and target places. Note that the shown markings represent the state after execution of the Transformation Net.

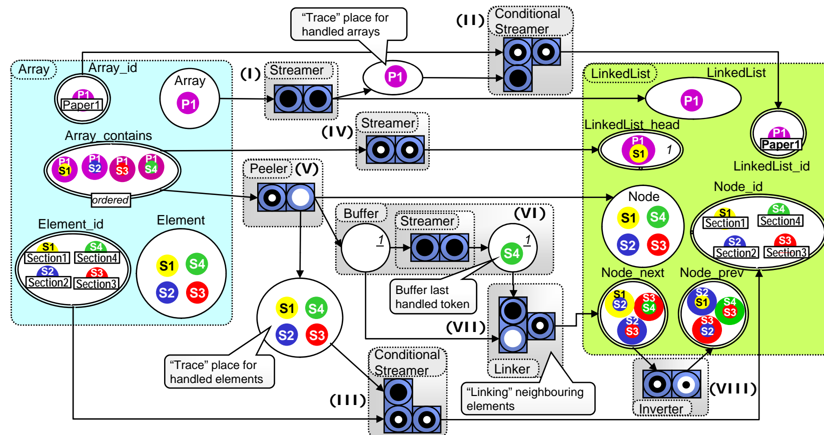

Fig. 7. Motivating Example solved with Transformation Nets

Starting from the top, we see `Transition (I)` that matches one-colored tokens from the "Array" place and propagates them into the "LinkedList" place using the `Streamer` pattern. The "id" attributes are streamed once their owning objects have been transformed, which is the case when the owning objects are present in the nets "trace" places by applying `ConditionalStreamers`, namely `Transition (II)` and `Transition (III)`. Analogously, the `Transition (IV)` matches two-colored tokens from the "contains" place and propagates them to the "head" place. Note that this transition can only fire once, because the "head" place has a relative capacity of one, only. Due to the fact that the input place is

of the ordered kind, the transition matches the tokens starting from the "zero-th" link of the reference. The third and final case where such primitive transition logic suffices is the transformation of all "Element" tokens to the "Node" place. Compared to that, dealing with "contains" tokens is of more interest. Thereby, two-colored tokens are matched from the "contains" place, and the adjacent `Transition (V)` peels out the outer ring, basically producing a duplicate of an "Element" token. The outcome is then put into a place with absolute capacity of one to enforce the ordering of tokens. Only if this place is empty again, the transition matching "Array_contains" tokens can produce the follow-up token therein. That place is cleared by the switching of the adjacent `Transition (VI)`, which consumes its input and moves it to its output with an absolute of capacity one. In case both places are filled with a one-colored token, the `Transition (VII)` is enabled which produces a two-colored token out of its inputs that is streamed into the "next" place. When firing, only the token from the right-most place is consumed, thus freeing the place to be filled again. Hence, this Transformation Net pattern forms a buffer-like structure of two places, which are, once they are filled, partly emptied to make space for successor tokens in the "queue" of places with single capacity. `Transition (VIII)` inverts and copies the created two-colored tokens from the "Node_next" to the "Node_prev" place.

The example has been realized with the help of our prototype tool supporting modeling, executing, and debugging Transformation Nets. Further details about the tool support may be found at our project page[4]. In order to show that Transformations Nets hide complexity in contrast to CPNs Fig. 8 depicts an equivalent CPN executing the same transformation. Firstly, to express the fact that tokens are not consumed per default, tokens consumed from a place are streamed back again. In order to avoid multiple firings every token gets an index and an additional place storing a counter is added to a transition. The transitions uses this counter to match for a specific token which is incremented after firing to match for the next available token (see label (I) in Figure 8). Secondly, standard CPNs offer no built-in concepts to express absolute and relative capacities. For absolute capacities we therefore again add an additional place, cf. CPN patterns presented in [11], holding the required number of tokens (see label (II) in Figure 8). Relative capacities can be expressed using a complex arc inscription, labeled (III) in Figure 8. Thereby it is shown that Transformation Nets can be mapped to standard CPNs in order to make use of already existing, efficient execution engines or to apply formal CPN properties in order to check the specification of the Transformation Net.

5 Lessons Learned

This section presents lessons learned from the running example and thereby discusses key features of the Transformation Net approach.

Representation of model elements by colored tokens reveals traceability. The source model to be transformed is represented by means of one-

[4] http://www.modeltransformation.net

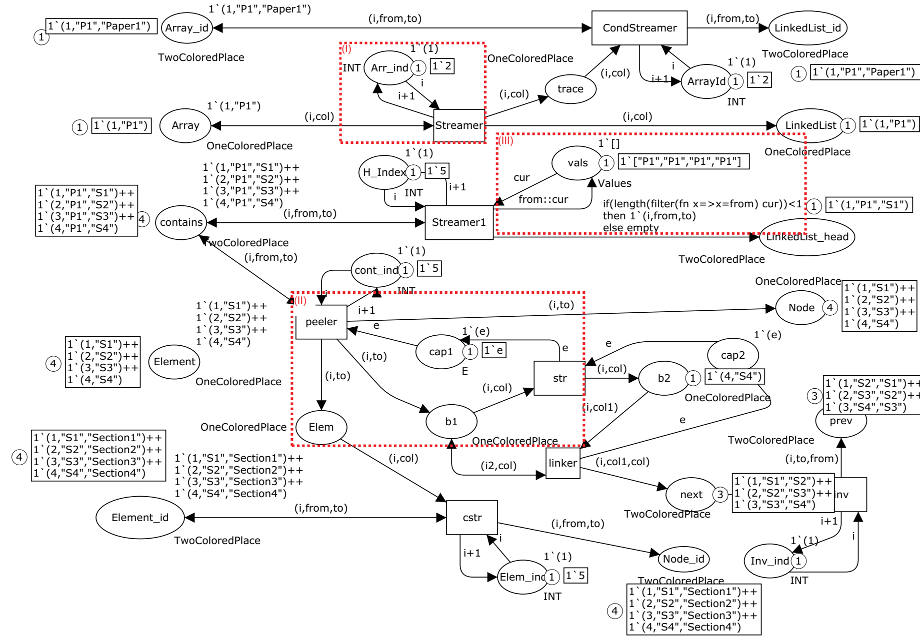

Fig. 8. Array2LinkedList using CPN

colored tokens and two-colored tokens residing in the source places of the Transformation Net whereby the actual transformation is performed by streaming these tokens to the target places. Through this mechanism it is possible to derive the source-target relationship, i.e., traceability, between model elements by searching for same-colored tokens in source places and target places, respectively.

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, 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 Transformation 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. Due to this fine grained decomposition we can not ensure a target model that is conform to its metamodel during transformations. The conformance to the target metamodel is checked when the target model is instantiated using the tokens in the target places.

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 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

Concerning our Transformation Net approach, we consider two orthogonal threads of related work. First, we discuss current model transformation approaches and point out their debugging support, and second, we elaborate on the usage of Petri Nets for model transformations.

Model Transformation Languages. Model transformation languages in general can be divided into imperative, declarative and hybrid approaches. Basically, imperative approaches are similar to direct manipulation of models through some general purpose programming language API, but offer a dedicated concrete syntax and allow to define the transformation in terms of the models abstract syntax. The operational part of the QVT specification [2] allows to define mappings, which are function-like constructs that can be imperatively called to create target elements. Declarative approaches are typically based on defining rules that are later on interpreted by an execution engine to produce the desired result. Hence, the actual transformation execution as well as the order of rule application generally need not be handled by the user. The way how a transformation is defined, is by specifying rules that constrain under which condition certain elements of the source and target metamodel are related. One part of the QVT specification consists of the Relations language, which allows to define rules in the above described way. However, what a declarative approach gains in abstraction, it loses in flexibility. To alleviate these limitations, hybrid approaches

combine imperative and declarative styles of transformation definition. The Atlas Transformation Language (ATL) [9] is a QVT-like language that distinguishes between declaratively matched and imperatively called rules.

However, the benefits of an explicit runtime model for the execution is not considered by these approaches. Instead low-level execution engines are employed which aggravates debugging and understanding of model transformation. By the process-oriented specification and execution of model transformations using Transformation Nets, we combine the benefits of imperative and declarative approaches not only for the specification of transformations, but also for their execution by using CPNs themselves as a transformation engine, which is currently not supported by hybrid approaches.

Petri Nets employed for Model Transformations. The relatedness of Petri nets and graph rewriting systems has induced some impact in the field of model transformations. 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 [13] 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 totally different. While these two approaches are using Petri nets as a back-end for automatically analyzing properties of transformations by employing place/transition nets, we are using a variant of CPNs to specify transformations and to foster debuggability and understandability of transformations. 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 Conclusions and Future Work

In this paper we showed how Transformation Nets, which are a variant of CPNs, can be used to specify and execute model transformations. Thereby we showed how metamodels, models and transformation logic can be expressed in Transformation Nets, providing an integrated view on all transformation artifacts involved as well as a dedicated runtime model to foster debugging. Finally we showed how concepts of Transformations Nets could be expressed in terms of CPNs.

Currently, we are working on a automated translation of Transformation Nets to standard Colored Petri Nets in order to make use of efficient execution engines of third party vendors. This furthermore allows us to use Petri net properties for analyzing and verifying model transformations which is another direction we are going to strive for future work, e.g., liveness properties to check if a transformation finishes.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments.

References

1. Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Core specification, 2006.
2. Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification, 2007.
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 @ MODELS'05*, 2005.
10. F. Legler and F. Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. *12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW'07)*, 2007.
11. N. Mulyar and W. M. von der Aalst. Towards a pattern language for colored petri nets. In K. Jensen, editor, *Proceedings of Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 39–59, Aarhus, Denmark, 2005.
12. T. Reiter, M. Wimmer, and H. Kargl. Towards a runtime model based on colored Petri-nets for the execution of model transformations. *3rd Workshop on Models and Aspects @ ECOOP'07*, 2007.
13. 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.