Taming the Shrew – Resolving Structural Heterogeneities with Hierarchical CPNs*

M. Wimmer[1], G. Kappel[1], A. Kusel[2],
W. Retschitzegger[2], J. Schoenboeck[1], and W. Schwinger[2]

[1] Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
{firstname.lastname}@jku.at

Abstract. Model transformations play a key role in the vision of Model-Driven Engineering (MDE) whereby the overcoming of structural heterogeneities, being a result of applying different meta-modeling constructs for the same semantic concept, is a challenging, recurring problem, urgently demanding for reuse of transformations. In this respect, an approach is required which (i) abstracts from the concrete execution language allowing to focus on the resolution of structural heterogeneities, (ii) keeps the impedance mismatch between specification and execution low enabling seamless debuggability, and (iii) provides formal underpinnings enabling model checking. Therefore, we propose to specify model transformations by applying a set of abstract mapping operators (MOPs), each resolving a certain kind of structural heterogeneity. For specifying the operational semantics of the MOPs, we propose to use Transformation Nets (TNs), a DSL on top of Colored Petri Nets (CPNs), since it allows (i) to keep the impedance mismatch between specification and execution low and (ii) to analyze model transformations by evaluating behavioral properties of CPNs.

Key words: Model Transformation Reuse, Hierarchical CPNs, Structural Heterogeneities, Mapping

1 Introduction

MDE is a current trend in software engineering where models are used as first-class artifacts throughout the software lifecycle [2], which are then systematically transformed to concrete implementations. In this respect, model transformations play a vital role, representing *the* key mechanism for *vertical transformations* like the generation of code and *horizontal transformations* like model exchange between different modeling tools, to mention just a few. In the context of transformations between different metamodels and their corresponding

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

models, the overcoming of *structural heterogeneities*, being a result of applying different meta-modeling constructs for the same semantic concept [11, 13] is a challenging, recurring problem, urgently demanding for reuse of transformations.

In this respect, reusable transformations should abstract from a concrete transformation language, allowing to (preferably graphically) specify transformations in an explicit *specification view* without having to struggle with the intricacies of a certain transformation language. Secondly, for being able to debug and comprehend resulting specifications, the impedance mismatch between the specification view and the executable formalism needs to be minimized, demanding for a *debugging view* which retains the structure of the specification view, i.e., components used in the specification view should not get scattered in the debugging view. Finally, since debugging can only provide limited evidence of correctness by means of a set of test runs, the underlying executable formalism for the *execution view* should provide means to enable *model checking* [3].

We therefore propose to specify horizontal model transformations by means of abstract mappings representing a set of reusable transformation components, called mapping operators (MOPs), to resolve recurring structural heterogeneities. These MOPs operate on different levels of granularity, i.e., we provide a set of *kernel MOPs* representing the basic functionality needed for resolving structural heterogeneities and a set of *composite MOPs* encapsulating several kernel MOPs, thus enhancing scalability of our approach. In order to specify the operational semantics of the MOPs, we propose to use TNs [23], a DSL on top of CPNs [9], since TNs allow to keep the impedance mismatch between specification view and debugging view low by encapsulating the transformation logic of a single MOP together with the metamodels and the models. Thereby debuggability and comprehensibility are fostered, i.e., the ability of finding and reducing the number of bugs. Moreover, the underlying CPNs allow to specify reusable components in the form of modules, which can be nested in a hierarchical way, allowing to accordingly represent composite MOPs. Therefore the main contribution of this paper is to enable reuse also on the execution level, i.e., the Petri Net layer. Finally, the formal underpinnings of CPNs allow the application of generally accepted behavioral properties to analyze the transformation specification. The whole framework is called TROPIC – TRansformations On Petri nets In Color.

The remainder of this paper is structured as follows. Section 2 introduces a motivating example, Section 3 concentrates on the specification of a transformation and Section 4 deals with the debugging thereof. The subsequent Section 5 shows how TNs are represented in standard CPNs and how behavioral properties are exploited to analyze the transformation specification. Lessons learned are discussed in Section 6 and related work is surveyed in Section 7. Finally, Section 8 concludes the paper with an outlook on future work.

2 Motivating Example

Structural heterogeneities between different metamodels occur due to the fact that semantically equivalent concepts can be expressed by different metamodeling concepts, e.g., explicitly by classes or only by attributes. Fig. 1 shows an

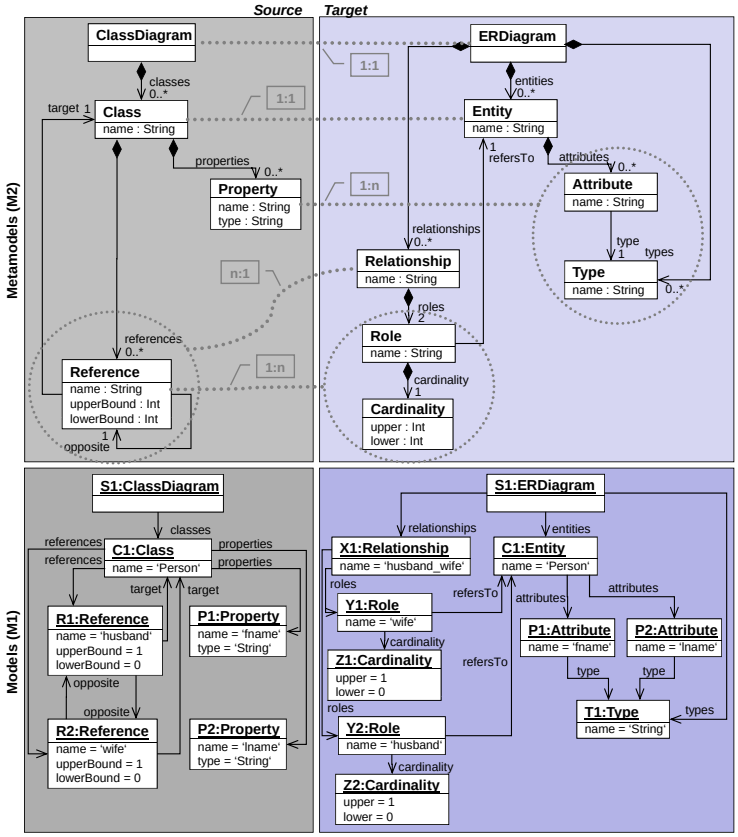


Fig. 1. Metamodels and Models of the Running Example

example used throughout the rest of the paper which exhibits common structural heterogeneities between metamodels, applying different modeling constructs to represent relationships as can be found e.g., in Ecore[3] or in Entity-Relationship Models. The `ClassDiagram` shown on the left side of Fig. 1, only provides unidirectional references, thus bidirectionality needs to be modeled by a pair of opposite references. In contrast to that, the `ERDiagram` explicitly represents bidirectionality, allowing to express relationships in more detail, e.g., using roles.

In the following, the main correspondences between the `ClassDiagram` and the `ERDiagram` are shortly described. On the level of classes, three main correspondences can be recognized, namely *1:1 correspondences*, *1:n correspondences* and *n:1 correspondences*, which are also indicated by dotted lines in Fig. 1. 1:1 correspondences can be found (i) between the root classes `ClassDiagram` and `ERDiagram` as well as (ii) between `Class` and `Entity`. Regarding 1:n correspondences, again two cases can be detected, namely (i) between the class `Property` and the classes `Attribute` and `Type` and (ii) between the class `Reference` and the classes `Role` and `Cardinality`. Although these are two occurrences of a 1:n

[3] http://www.eclipse.org/modeling/emf/

correspondence, there is a slight difference between them, since in the first case only for distinct values of the attribute `Property.type`, an instance of the class `Type` should be generated. Finally, there is one occurrence of a n:1 correspondence, namely between the class `Reference` and the class `Relationship`. It is classified as n:1 correspondence, since for *every pair* of `References`, that are opposite to each other, a corresponding `Relationship` has to be established. Considering attributes, only 1:1 correspondences occur, e.g., between `Class.name` and `Entity.name`, whereas regarding references, 1:1 correspondences and 0:1 correspondences can be detected. Concerning the first category, one example thereof arises between `ClassDiagram.classes` and `ERDiagram.entities`. Regarding the latter category, e.g., the relationship `ERDiagram.types` exists in the target without any corresponding counterpart in the source.

3 Specification View

As mentioned before, the actual specification of a transformation problem should abstract from a concrete transformation language allowing the transformation designer to focus on the resolution of structural heterogeneities without having to struggle with the intricacies of a certain transformation language. Therefore we propose to specify model transformations by means of abstract mappings being a declarative description of the transformation, as known from the area of data engineering [1]. For this we provide a library of composite MOPs [21]. Thereby we identified typical mapping situations being 1:1 copying, 1:n partitioning, n:1 merging, and 0:1 generating of objects, for which different MOPs are provided. In this respect, reuse is leveraged as the proposed MOPs are generic in the sense that they abstract from concrete metamodel types since they are typed by the core concepts of current meta-modeling languages like Ecore or MOF (i.e., class, attributes, references). To further structure the mapping process we propose to specify mappings in two steps.

In a first step, *composite MOPs*, describing mappings between classes are applied, providing an abstract *blackbox-view* (cf. Fig. 2). Every composite MOP consists of so-called *kernel MOPs*, thus the composite behavior is realized by a set of basic building blocks. These kernel MOPs are responsible for resolving structural heterogeneities and therefore they have to be able to map classes, attributes, and references in all possible combinations and mapping cardinalities. In this respect, MOPs are provided for copying exactly one object, value, or link from source to target, respectively (denoted as C(lass)$_2$C(lass), A(ttribute)$_2$A(ttribute), and R(eference)$_2$R(eference)). Moreover, MOPs are needed for merging objects, values, and links (denoted as C_2^nC, A_2^nA, and R_2^nR) resolving the structural heterogeneity that concepts in the source metamodel are more fine-grained than in the target metamodel. Finally, MOPs are needed for generating a target element without an obvious source element (denoted as 0_2C, 0_2A, and 0_2R) to resolve heterogeneities resulting from expressing the same modeling concept with different meta-modeling concepts – a situation which often occurs in metamod-

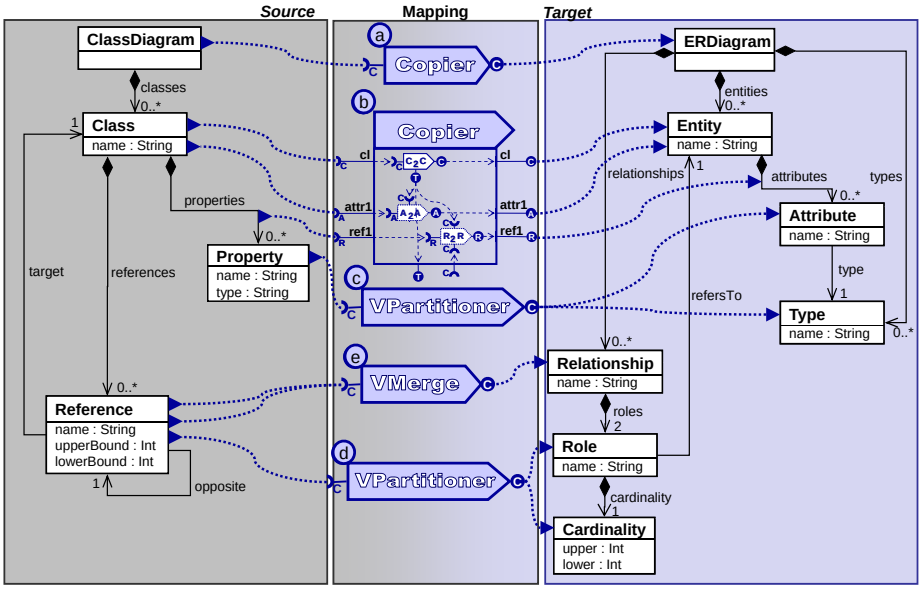


Fig. 2. Solution of the Running Example

eling practice.[4] In a second step, the composite MOPs, which solely describe a mapping between classes at first, have to be refined to also map attributes and references in the so-called *whitebox-view* by the usage of kernel MOPs (cf. expanded Copier (b) in Fig. 2). Furthermore, kernel MOPs can be used to assemble new, user-defined composite MOPs.

As a concrete syntax for MOPs we are using a subset of the UML 2 component diagram concepts enabling the specification of model transformations in a plug & play manner. With this formalism, every MOP is defined as a dedicated component, representing a modular part of the transformation specification which encapsulates an arbitrary complex structure and behavior, providing well-defined interfaces. Every MOP has input ports with required interfaces (left side of the component) as well as output ports with provided interfaces (right side of the component), typed to classes (C), attributes (A), and relationships (R) (cf. `Copier` (b) in Fig. 2). Since there are dependencies between MOPs, e.g., a value can only be set after the owning object has been created, MOPs dealing with the transformations of classes additionally offer a trace port (T) at the bottom providing *context information*, indicating which target object has been produced from which source object(s). This port can be used by dependent MOPs to access context information via required context ports (T). In case of MOPs dealing with the mapping of attributes the corresponding interface is shown via one port on top, or in case of MOPs dealing with the mapping of ref-

[4] Please note, that although composite MOPs for 1:n partitioning are provided, no additional kernel MOps are needed, since such situations can be simulated by n x 1:1 MOps.

Table 1. Overview of Composite MOPs used in the Example

Correspondence	MOP	Description	Composition of Kernel MOPs (EBNF)
1:1 - copying	Copier	creates exactly one target object per source object	Copier: C_2C { A_2A | $A^n{}_2A$ | 0_2A | R_2R | $R^n{}_2R$ | 0_2R }
1:n - partitioning	VerticalPartitioner	splits one source object into several target objects	VerticalPartitioner: Copier { ObjectGenerator | Copier }
n:1 - merging	VerticalMerger	merges several source objects to one target object	VerticalMerger: $C^n{}_2C$ { A_2A | $A^n{}_2A$ | 0_2A |R_2R | $R^n{}_2R$ | 0_2R }
0:1 - generating	ObjectGenerator	generates a new target object without corresponding source object	ObjectGenerator: 0_2C { A_2A | $A^n{}_2A$ | 0_2A | R_2R | $R^n{}_2R$ | 0_2R }

erences via two ports, whereby the top port depicts the required source context and the bottom port the required target context (cf. `Copier` (b) in Fig. 2).

For solving the running example, several composite MOPs have been applied as can be seen in Fig. 2. Table 1 presents an overview of the used composite MOPs to solve the example as well as their composition of kernel MOPs. For a detailed classification and description of all available kernel as well as composite MOPs we refer to [21]. To resolve the 1:1 correspondences between `ClassDiagram` and `ERDiagram` as well as between `Class` and `Entity` in our example we applied two `Copiers` since for every source object a corresponding target object should be generated (cf. MOPs (a) and (b) in Fig. 2)). The whitebox-view of the `Copier` (b) thereby shows the mapping of class `Class` to class `Entity` using a C_2C MOP. Moreover, the attribute `Class.name` is mapped to the attribute `Entity.name` by using an A_2A MOP. Finally, the reference `Class.properties` is mapped to the reference `Entity.attribute` using a R_2R MOP. To split the attributes of the class `Reference` to the target classes `Role` and `Cardinality` a `VerticalPartitioner` is applied (cf. MOP (d) in Fig. 2). Besides this default behavior, aggregation functionality is sometimes needed as is the case when splitting the `Property` concept into the `Attribute` and `Type` concepts, since a `Type` should only be instantiated for distinct `Property.type` values (cf. MOP (c) in Fig. 2). To merge two `Reference` objects to a single `Relationship` object a `VerticalMerger` is applied (cf. MOP (e) in Fig. 2).

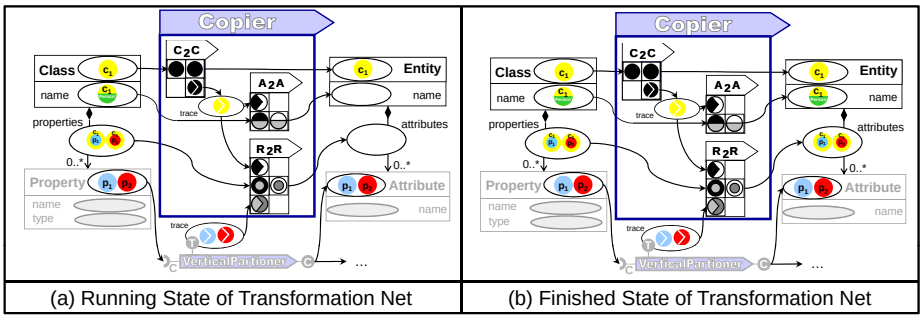
4 Debugging View

In the previous section we showed how structural heterogeneities can be resolved by applying MOPs resulting in a declarative mapping specification. In order to execute this specification it has to be translated into an executable formalism, i.e., every MOP has to be assigned an operational semantics. Thereby, the impedance mismatch between the declarative specification and the actual operational semantics should be minimized in order to foster comprehensibility and debuggability. Since current transformation languages (cf. [4] for an overview) provide only a limited view on a model transformation problem, i.e., they do not visualize the actual metamodel and model being transformed, we proposed the TN formalism [23], being a DSL on top of CPNs [9]. The basic idea of TNs is to represent the transformation logic together with the metamodels and the models, whereby metamodel elements are represented by places, model elements by the according markings and the actual transformation logic by a system of transitions. Thus, an explicit runtime model is provided which can be used to observe the runtime behavior of a certain transformation. In the following we describe

the core concepts of TNs as well as the adaptations introduced in comparison to standard CPNs to better suit the domain of model transformations.

Representation of Metamodels and Models. Since we rely on the core concepts of an object-oriented meta-metamodel the graph which represents the metamodel consists of classes, attributes, and references which are represented by according places in TNs. Therefore Fig. 3 depicts a place for the class `Class` as well as one place for the attribute `Class.name` and one place for the reference `Class.properties`. The graph which represents a conforming model consists of objects, data values and links which are represented by tokens in the according places. For every object that occurs in a model a one-colored *ObjectToken* is produced, which is put into a place that corresponds to the respective class in the source metamodel, e.g., the token `C1` in the `Class` place and the tokens `P1` and `P2` in the place `Property`, representing the objects of the source model depicted at the bottom of Fig. 1. The color is realized through a unique value that is derived from the object id (OID). For every value, two-colored *AttributeTokens* are produced whereby the upper color represents the object and the lower color the actual value, e.g., the `C1|Person` token represents the value "Person" of the attribute `Class.name` for the object `C1` in Fig. 3. Finally, for every link a two-colored *ReferenceToken* is produced. The outer color refers to the color of the token that corresponds to the owning object. The inner color is given by the color of the token that corresponds to the referenced target object, which is depicted by the corresponding tokens in the `Class.properties` place in Fig. 3.

Specification of Transformation Logic. The actual transformation logic is specified by means of a system of transitions and additional places, so-called *trace places* storing *context information* which reside in-between those places representing the original input and output metamodels. Transitions consist of so-called *query tokens* (LHS of the transition) representing the pre-condition of a certain transition, whereas *production tokens* (RHS of the transition) depict its postcondition. Thereby different query and production tokens for objects, values, links and context information are provided whose colors represent variables that are bound during execution, i.e, colors of query tokens are not the required colors for input tokens, instead they describe configurations that have to be fulfilled by input tokens. In the copying sceanrio the color of the production tokens depend on the color of the query tokens, e.g., the production token and the query token



(a) Running State of Transformation Net (b) Finished State of Transformation Net

Fig. 3. Debugging View of Copier MOP

of the C_2C transition exhibit the same color and therefore the source and the target object tokens exhibit the same color (cf. Fig. 3). However, it is also possible to produce a token of a not yet existing color if a target object is needed which does not directly correspond to a source object, e.g., in case a C_2^nC MOP which merges several source objects to a single new target object. Furthermore, to represent trace ports of MOPs, *trace places* containing *context tokens* indicate which target object has been created from which source object(s). Thereby the color(s) of the slot (left side) indicate(s) the used source object(s) whereas the generated target object is represented by the color of the remaining slice (right side of token). Since object tokens are simply copied in case of the depicted C_2C transition source and target context tokens exhibit equal colors (cf. Fig. 3(a)). Only if context information is available in a trace place, dependent transitions, e.g., the A_2A and R_2R transitions, are able to fire. For this, they query the context tokens in order to add a value or a link to the target object acquired from the context token (cf. Fig. 3(b)). Please note that in case of creating a new target object, source and target color of the context tokens differ from each other. Thus, dependent transitions must be able to cope with differently colored context tokens and therefore the context query tokens of the dependent A_2A and R_2R transitions in Fig. 3 show different colors (which are only variables and are therefore also able to match for same colored tokens).

Adaptations of Standard CPNs. In contrast to standard CPNs, TNs exhibit a different default firing behavior, i.e., tokens are not consumed per default (therefore source tokens are preserved in their corresponding source places). This is since all possible token combinations must be taken into account. For example, if the R_2R transition would consume `Class` tokens and `Property` tokens from the trace places (cf. Fig. 3), the transition could fire only once although multiple `Properties` would be available, since there is a 1:n relationship between `Class` and `Property`. Moreover, if more than one transition accesses a certain place, consuming firing behavior would lead to erroneous race conditions.

Summarizing, TNs provide a formalism to specify the operational semantics of the provided MOPs. Thereby TNs reduce the impedance mismatch between the abstract declarative mapping specification and the actual operational semantics since there is a 1:1 correspondence between kernel MOPs and transitions. Additionally, all artifacts in a model transformation, i.e., metamodel, transformation logic and the involved models are represented in a homogenous view. Furthermore, as query and production tokens are only typed to the core concepts of object-oriented metamodels (class, attributes and references) the specified transformation logic can be reused between arbitrary metamodels (as intended by the MOPs). Due to the fact that every MOP is realized by an independent set of transitions every MOP can be debugged individually, thus enabling a component-oriented debugging approach.

5 Execution View

Since TNs represent a DSL on top of CPNs they can be fully translated into existing CPN concepts to make use of efficient execution engines and their properties

to analyze model transformations [20]. The actual translation is transparent to the user since a TN is automatically converted to an according CPN using the ASAP platform [19]. The ASAP platform provides an EMF-based implementation of the PNML standard[5] for CPNs. The CPN model can then be used to check the syntax of the corresponding TN, to simulate the TN and to calculate behavioral properties for the specified model transformations. Since every MOP is realized by an independent set of TN transitions we provide pre-defined hierarchical CPNs for kernel and composite MOPs, detailed in the following. Furthermore, the application of behavioral properties for analyzing model transformations is shown.

5.1 Representation of Kernel MOPs

Kernel MOPs and their respective operational semantics in TNs can be represented by means of *modules* or so-called *substitution transitions* in hierarchical CPNs whereby the *ports* of the substitution transitions are only typed by classes, attributes, and references. The ports are then bound to the corresponding *socket places* being the places derived from the source and target metamodel. In the following we show how to realize the non-consuming behavior in CPNs as well as the translation of kernel MOPs to hierarchical CPNs.

Adaptations of Standard CPNs. To realize the non-consuming firing behavior, a so-called *history place* is introduced for every transition. It stores all token combinations that have already been fired by this transition in a sorted list in order not to blow up the state space, i.e., there is no difference if token `P1` or token `P2` has been transformed first in our scenario. The history place is connected to the corresponding transition whereby a guard condition prevents the transition from firing a certain token combination twice. Moreover, the standard arcs are replaced by so-called *test arcs*, which do not consume tokens from the connected input places. For further details on the translation of TNs to CPNs we refer the interested reader to [23].

MOPs mapping Classes. In case of kernel MOPs dealing with the mapping of classes, e.g., a C_2C MOP as depicted in Fig. 4(a), the in- and outports have to be typed to the colorset `Class` (`colset Class = record object : INT * name : STRING`). As a C_2C MOP simply copy tokens, the same arc inscription can be found on the in- and outgoing arcs (represented by the same colors of query and production token in TNs). Furthermore, kernel MOPs mapping classes provide context information stored in the *context* port[6]. The colorset `Context` thereby defines a record consisting of a list of classes (since more than one class can be used to enable the transition in case of a C_2^nC) and a target class (`colset Context = record source:SourceContext * target : Class; colset SourceContext = list Class;`).

[5] http://www.pnml.org/

[6] Note that ports providing context information in MOPs and CPNs are used to enable dependent MOPs or transitions, i.e, they provide required tokens to enable a transition, whereas the history concepts solely hinders multiple firings of transition in CPNs

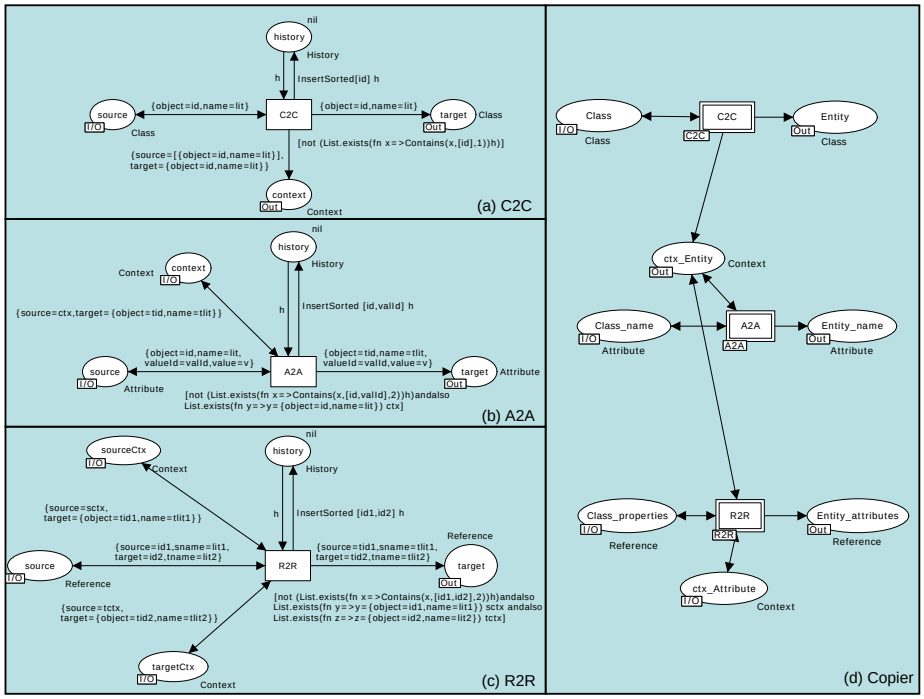


Fig. 4. Realization of MOPs by Hierarchical CPNs

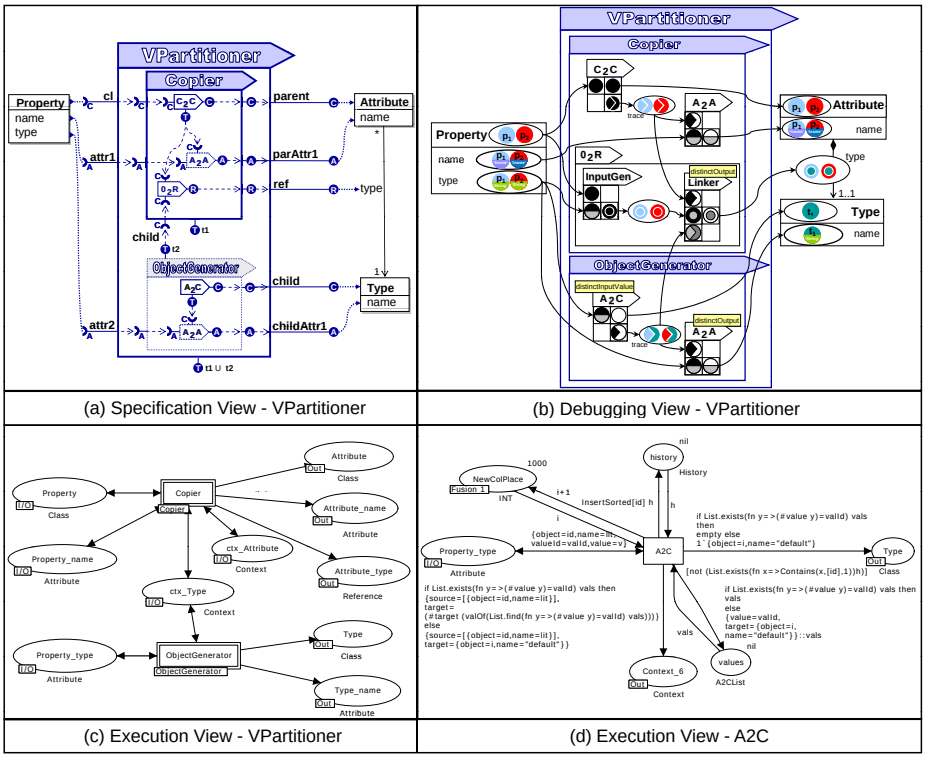
MOPs mapping Attributes or References. In case of kernel MOPs dealing with the mapping of attributes or references, e.g., an A_2A MOP or R_2R MOP as depicted in Fig. 4(b) and (c), the ports have to be typed to the colorset `Attribute` and `Reference` respectively (`colset Attribute = record object:INT * name:STRING * valueId : INT * value : STRING; colset Reference = record source:INT * sname: STRING * target:INT * tname: STRING;`). Since attributes and references should only be transformed if the owning object of an attribute or the source and target objects of a reference have already been transformed, the guard condition of the transition not only prevents the multiple firing but additionally checks if the context place already contains the necessary context information. If the condition is fulfilled, an attribute or reference token is produced whereby the new owning object `tid` is acquired from the context tokens (cf. arc inscription at the in- and outgoing arcs from context places Fig. 4(b) and (c)). These hierarchical CPNs can then be assembled to more coarse-grained hierarchical CPNs to represent, e.g., a `Copier` as shown in Fig. 4(d). In the following, composite MOPs are elaborated in more detail.

5.2 Representation of Composite MOPs

Specification View. In Section 3 we introduced coarse-grained composite MOPs which encapsulate several kernel MOPs, e.g., a `Copier` consists of exactly one C_2C, and several MOPS for mapping attributes or references. As can be seen in

Table 1, composite MOPs can not only consist of kernel MOPs but might encompass composite ones themselves, e.g., `VerticalPartitioner` which consists of a `Copier` and an `ObjectGenerator` (cf. Fig. 5(a)). In our running example this MOP was used to split the source concept `Property` into the concepts `Attribute` (achieved by the contained `Copier`) and `Type`, whereby a `Type` should only be instantiated for distinct `Property.type` values overcoming the heterogeneity that a concept is expressed as an attribute in the source metamodel and as a class in the target metamodel (achieved by the contained `ObjectGenerator`).

Debugging View. The relation between composite and the kernel MOPs can be seen in the debugging view (cf. Fig. 5(b)). First, the C_2C transition of the copier streams the corresponding object tokens, thus creating an `Attribute` for every `Property`. The thereby generated context information enables the A_2A transition in order to set the `Attribute.name` values. Second, the A_2C transition generates a `Type` object token for distinct `Property.type` values, which is indicated by the `distinctInputValue` annotation on the transition meaning that only context information in the according trace place is generated but no new target token in case that a value occurs several times. Therefore, the trace place of the `ObjectGenerator` composite MOP contains two `Property.type` tokens which both have been mapped to the same `Type` object (depicted by the equal target color of the context tokens) since both source tokens have the same



(a) Specification View - VPartitioner

(b) Debugging View - VPartitioner

(c) Execution View - VPartitioner

(d) Execution View - A2C

Fig. 5. Different Views of VerticalPartitioner

value ``String''. In order not to produce too many attribute tokens the dependent A_2A MOP has to match only for distinct target colors of context tokens resulting in distinct output values (indicated by the according annotation in (cf. Fig. 5(b)). Finally, the generated `Attribute` and `Type` objects have to be accordingly linked by the reference `Attribute.type`. Since there is no according source reference available we have to generate this reference by applying a 0_2R MOP. Nevertheless, the transformation designer has to define during specification how the generated target objects are related to each other in the source model. In our example the intention is to generate a reference for every `Attribute` object having set an according `Attribute.type` value. In order to get this input the `InputGen` transition collects the tokens and thereby generates (self) references. These references can then be processed by the `Linker` component which finally produces the according `Attribute.type` references.

Execution View. In order to represent the different levels of granularity, the corresponding hierarchical CPN again consists of several nested ones, thus leading to multi-level hierarchical CPNs. As shown in Fig. 5(c) the `VerticalPartitioner` consists of two substitution transitions, being a `Copier` and an `ObjectGenerator`. As already shown in the debugging view (cf. Fig. 5(b)), the main part of an `ObjectGenerator` is an A_2C kernel MOP. Since only for distinct values a new target object should be generated, an additional `values` place containing a list of records (`colset A2CList = list A2C; colset A2C = record value:INT * target:Class`), expressing which values have already been converted to a certain class token, is introduced (cf. Fig. 5(d)). The conditions on the outgoing arcs to the target place and to the `values` places ensure that a token is only created if the value has not been contained in the `values` list before. In contrast to that, context information is produced for any firing of the transition whereby the source object is connected to an already existing class target token if the values list contains an according entry, i.e., if a `Type` has already been created for a certain `Property.type` value. To represent the fact that a `Type` object has no according counterpart in the source model, we generate a new object id which is the task of the (fusion) place `NewColPlace` and the according arc inscriptions represented by a newly colored object production token in TNs.

5.3 Behavioral Properties to Analyze Mappings

Although the operational semantics of MOPs is predefined, configuration errors might occur when applying the MOPs in the specification phase leading to an erroneous interplay between MOPs. In the following we show how typical errors can be detected by means of behavioral properties of the underlying CPNs [20].

Model Comparison using Boundedness Properties. Typically, the first step in analyzing the correctness of a transformation is to compare the generated target model to an expected target model. To identify wrong or missing target elements in terms of tokens automatically, *Boundedness* properties can be applied. An example thereof could be the A_2C MOP in the above example which creates target tokens for distinct values only. Therefore dependent transitions need to generate a distinct output as well, e.g., to set the `Type.name` value only

once. If this is not specified by the user accordingly, too many `Type.name` tokens are generated which can be detected by comparing the Boundedness properties of the according place of the generated model to the expected target model.

Checking Interplay of MOPs using Liveness Properties. Another source of error during the refinement of composite MOPs by kernel MOPs is the mapping of dependent attributes and references. In case that MOPs dealing with attributes and references are connected to wrong source or target context ports the corresponding transition is not able to fire which can be detected by *Liveness Properties* such as *Dead Transition Instances* or *L0-Liveness*.

Termination and Confluence Analysis using Dead and Home Markings. A transformation specification must always terminate, thus the state space has to contain at least one *Dead Marking*, which is typically ensured by the history concept. Moreover, it has to be ensured that a dead marking is always reachable, meaning that a transformation specification is confluent, which can be checked by the *Home Marking*. Furthermore, it is possible to check if a certain marking, i.e., the target marking derived from the expected target model, is reachable. If this marking is equal to the Dead and Home Marking it is ensured that the specified mapping always generates the expected target model.

6 Lessons Learned

This section presents lessons learned and discusses key features of our approach.

Kernel MOPs Enable Extensibility. Kernel MOPs form the basis for overcoming structural heterogeneities and thereby have to exhibit a well-defined operational semantics. Since composite MOPs are solely based on kernel MOPs, the composite operational semantics results from the operational semantics of the kernel MOPs. Therefore, the library of composite MOPs can be easily extended on basis of the kernel MOPs without the need of adapting the compilation to TNs and CPNs, respectively.

CPNs Allow for Parallel Execution. As CPNs exhibit an inherent concurrency, parallel execution of transformation logic is possible thereby increasing the efficiency of a transformation execution. In particular, mappings between classes are independent from each other and therefore the transformation of objects can be fully parallelized. The same is true for depending attributes and references which can also be transformed in parallel after the owning objects have been created and thus the needed context tokens are available.

Visual Formalism Eases Debugging and Understandability. TNs provide a visual formalism for defining model transformations which is especially useful for debugging purposes, since the actual execution of a certain model transformation can be simulated. In this respect, the transformation of model elements can be directly followed by observing the flow of tokens and therefore undesired results can be detected easily.

History Ensures Termination. As mentioned above, TNs introduce a specific firing behavior in that transitions do not consume the source tokens satisfying the precondition but hold them in a history. Thus, a transition can only fire once for a specific combination of input tokens prohibiting infinite loops,

even for test arcs or cycles in the net. Only if a transition occurs in a cycle and if it produces new objects every time it fires, the history concept can not ensure termination. Such cycles, however, can be detected at design time and are automatically prevented for TNs. In contrast to model transformation languages based on graph grammars, where termination is undecidable in general [14], TNs ensure termination already at design time.

State Space Explosion Limits Model Size. A known problem of model checking and thus also of behavioral properties of Petri Nets is that the state space might become very large. Currently, the full occurrence graph is constructed to calculate properties leading to memory and performance problems for large source models and transformation specifications. Often a marking M has n concurrently enabled, different binding elements leading all to the same marking. Nevertheless, the enabled markings can be sorted in $n!$ ways, resulting in an explosion of the state space. As model transformations typically do not care about the order how certain elements are bound, the number of bindings can be reduced to 2^n bindings, thus enhancing scalability of our approach.

7 Related Work

In the following, related work is summarized according to the proposed views.

Specification View. In the area of model engineering only the ATLAS Model Weaver (AMW) [7] provides a dedicated mapping tool allowing the definition of model transformations independent of a concrete transformation language. By extending the weaving metamodel, one can define the abstract syntax of new weaving operators which roughly correspond to our MOPs. The semantics of weaving operators is determined by a higher-order transformation [16], taking a model transformation as input and generating another model transformation as output. Compared to our approach, the weaving models are compiled into low-level ATL [10] transformation code which is in fact a mixture of declarative and imperative language constructs. Thus, this solution exhibits an impedance mismatch, hindering the understanding and debugging of the resulting code.

Debugging View. Concerning model transformations in general, there is little debugging support available. Most often only low-level information available through the execution engine is provided, but traceability according to the higher-level correspondence specifications is missing. For example, in the Fujaba environment, a plugin called MoTE [18] compiles TGG rules [12] into Fujaba story diagrams that are implemented in Java, which obstructs a direct debugging on the level of TGG rules. In [8], the generated source code is annotated accordingly to allow the visualization of debugging information in the generated story diagrams, but not on the TGG level. Concerning the understandability of model transformations in terms of a visual representation and a possibility for a graphical simulation, only graph transformation approaches like 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 provide snapshots of the current transformation state, only.

Execution View. Current transformation languages provide only limited support to analyze transformation specifications as summarized in the following. In the area of graph transformations some work has been conducted that uses Petri Nets to check properties of graph production rules. Thereby, the approach proposed in [17] translates individual graph rules into a Place/Transition Net and checks for its termination. Another approach is described in [6], where the operational semantics of a visual language in the domain of production systems is described with graph transformations. The production system models as well as the graph transformations are transformed into Petri Nets in order to make use of analysis techniques for checking properties of the production system models. Finally, a recent work by de Lara and Guerra [5] proposes to translate QVT-Relations into CPNs - on the one hand to provide a formal semantics for QVT Relations and on the other hand to analyze QVT Relations specifications - pursuing similar ideas as followed in our previous work [22]. Nevertheless, these approaches are using Petri Nets only as a back-end for analyzing properties of transformations, whereas we are using a DSL on top of CPNs as a front-end for model transformations, thereby fostering debuggability.

8 Future Work

Currently, only the most important concepts of modeling languages, i.e., classes, attributes and relationships have been considered by our MOPs. It would be desirable, however, to extend our MOP library to be able to deal also with concepts such as inheritance or complex mathematical operations. Furthermore, only a basic prototype of the proposed debugging view is available. We therefore focus on improving our prototype, e.g., by accordingly visualizing the findings of the formal properties. Concerning verification support, we focused on small mapping scenarios up to now only, not least due to the state space explosion problem. Nevertheless the ASAP platforms provides the possibility to specify own algorithms to explore the state space which could additionally be adopted to the domain of model transformation to enable verification support for larger scenarios. To further support the transformation designer in complementing the mapping in the whitebox-view, auto-completion strategies should be incorporated. In this respect, we will investigate on matching strategies [15] which may be applied to automatically derive attribute and relationship mappings.

References

1. P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *Proc. of SIGMOD'07*, pages 1–12. ACM, 2007.
2. J. Bézivin. On the Unification Power of Models. *Journal on Software and Systems Modeling*, 4(2):31, 2005.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

4. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
5. J. de Lara and E. Guerra. Formal Support for QVT-Relations with Coloured Petri Nets. In *Proc. of MoDELS'09*, 2009.
6. J. de Lara and H. Vangheluwe. Automating the Transformation-Based Analysis of Visual Languages. *Formal Aspects of Computing*, 21, Mai 2009.
7. M. Del Fabro and P. Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *SoSyM*, 8(3):305–324, 2009.
8. L. Geiger. Model Level Debugging with Fujaba. In *Proceedings of 6th International Fujaba Days*, pages 23–28, Dresden, Germany, 2008.
9. K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modeling and Validation of Concurrent Systems*. Springer, 2009.
10. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.
11. V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: A context-based approach. *The VLDB Journal*, 5(4):276–304, 1996.
12. A. Koenigs. Model Transformation with TGGs. In *Proc. of Model Transformations in Practice Workshop of MoDELS'05*, Montego Bay, Jamaica, 2005.
13. F. Legler and F. Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. In *Proc. of BTW'07*, 2007.
14. D. Plump. Termination of graph rewriting is undecidable. *Fundamental Informatics*, 33(2), 1998.
15. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.
16. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *Proc. of ECMDA-FA'09*, pages 18–33, 2009.
17. D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformation by Petri Nets. In *Proc. of ICGT'06*, pages 260–274, 2006.
18. R. Wagner. Developing Model Transformations with Fujaba. In *Proc. of the 4th Int. Fujaba Days 2006*, pages 79–82, 2006.
19. M. Westergaard and L. M. Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In *Proc. of the 30th Int. Conf. on Applications and Theory of Petri Nets*, pages 313–322, 2009.
20. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger. Right or Wrong? - Verification of Model Transformations using Colored Petri Nets. In *Proc. of 9th DSM Workshop*, 2009.
21. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In *Proc. of ICMT'10*, 2010.
22. M. Wimmer, A. Kusel, J. Schoenboeck, G. Kappel, W. Retschitzegger, and W. Schwinger. Reviving QVT Relations: Model-based Debugging using Colored Petri Nets. In *Proc. of MoDELS'09*, pages 727–732, 2009.
23. M. Wimmer, A. Kusel, J. Schönböck, T. Reiter, W. Retschitzegger, and W. Schwinger. Lets's Play the Token Game – Model Transformations Powered By Transformation Nets. In *Proc. of PNSE'09*, pages 35–50, 2009.