Catch me if you can – Debugging Support for Model Transformations[⋆]

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

[1] Vienna University of Technology, Austria
{schoenboeck|kappel|wimmer}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
{kusel|retschitzegger}@bioinf.jku.at, wieland.schwinger@jku.ac.at

Abstract. Model-Driven Engineering places models as first-class artifacts throughout the software lifecycle requiring the availability of proper transformation languages. Although numerous approaches are available, they lack convenient facilities for supporting debugging and understanding of the transformation logic. This is because execution engines operate on a low level of abstraction, hide the operational semantics of a transformation, scatter metamodels, models, transformation logic, and trace information across different artifacts, and provide limited verification support. To tackle these problems, we propose a Domain-Specific Language (DSL) on top of Colored Petri Nets (CPNs)—called Transformation Nets—for the execution and debugging of model transformations on a high level of abstraction. This formalism makes the afore hidden operational semantics explicit by providing a runtime model in terms of places, transitions and tokens, integrating all artifacts involved into a homogenous view. Moreover, the formal underpinnings of CPNs enable comprehensive verification of model transformations.

Key words: Model Transformation, Debugging, CPN, Runtime Model

1 Introduction

The availability of model transformation languages is *the* crucial factor in MDE, since they are as important for MDE as compilers are for high-level programming languages. Several kinds of transformation languages have been proposed, comprising imperative, declarative, and hybrid ones [5]. Imperative approaches allow the specification of complex transformations more easily than declarative approaches, e.g., by providing explicit statefulness, but induce more overhead code as many issues have to be accomplished explicitly, e.g., specification of control flow. Although hybrid and declarative model transformation languages relieve transformation designers from these burdens, specification of transformation logic is still a tedious and error prone task due to the following reasons.

First, languages such as the Atlas Transformation Language (ATL) [10], Triple Graph Grammars (TGGs) [12], and QVT Relations [17] specify correspondences between source and target metamodel elements (cf. Fig. 1 (a)) on a

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

high level of abstraction, whereas accompanying execution engines operate on a considerably lower level. For example, ATL uses a stack machine and TGGs are first translated to Fujaba [21], and then to Java. As a consequence, debugging of model transformations is limited to the information provided by these engines, most often just consisting of variable values and logging messages, but missing important information, e.g., why certain parts of a transformation are actually executed. Thus, only a snapshot of the actual execution state is provided during debugging while coherence between the specified correspondences is lost. Therefore, these execution engines act as a black-box to the transformation designer hiding the operational semantics. Second, comprehensibility of transformation logic is further hampered as current transformation languages provide only a limited view on a model transformation problem. For example in ATL, metamodels, models, the transformation specification, and trace information are scattered across different artifacts. Graph transformation approaches using graph patterns also only reveal parts of the metamodel. Additionally, both approaches hide the transformation of concrete model elements. Finally, comprehensive verification support of model transformations is missing, although first approaches are available, e.g., work in the area of graph transformations such as [3].

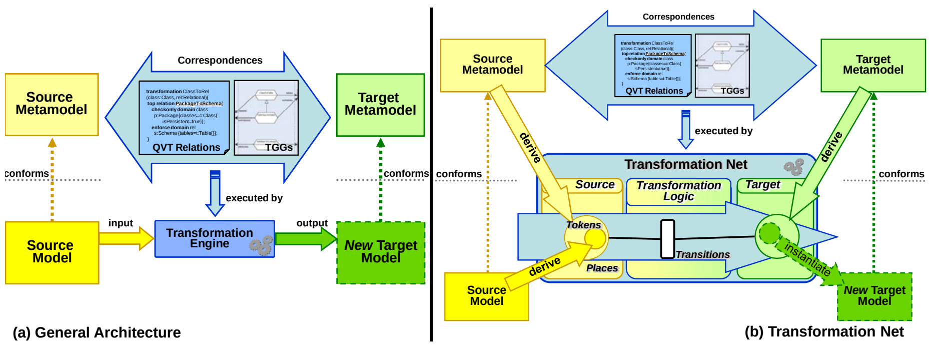


Fig. 1. Model Transformation Architecture

To tackle the aforementioned limitations of current approaches and their underlying execution engines we propose Transformation Nets (TNs), a DSL on top of Colored Petri Nets (CPNs) [9], for developing, executing and debugging model transformations (cf. Fig. 1(b)). In particular, for every metamodel element, places in TNs are derived, whereby a corresponding place is created for every class, every attribute and every reference. Model elements are represented by tokens which are put into the according places. Finally, the actual transformation logic is represented by transitions. The existence of certain model elements (i.e., tokens) allows transitions to fire and thus stream these tokens from source places to target places representing instances of the target metamodel to be created. This approach follows a process-oriented view towards model transformations allowing debugging on an appropriate level of abstraction. Furthermore, TNs provide the explicit statefulness of imperative approaches through tokens contained within places. The abstraction of control flow known from declarative approaches is achieved as the net's transitions can fire autonomously, thus making use of implicit, data-driven control flow. Nevertheless, the stepwise firing of the transitions makes explicit the operational semantics of the transformation logic and thereby enables *simulation-based debugging* (cf. (1) in Fig. 2). The ability to combine all the artifacts involved, i.e., metamodels, models, as well as the actual transformation logic, into a single representation makes the formalism especially suited for gaining an understanding of the intricacies of a specific model transformation. Moreover, TNs form a runtime model, serving as an execution engine for diverse model transformation languages, e.g., QVT Relations. As the runtime model itself is formally specified in terms of a metamodel [24], it can be exploited for *model-based debugging* by using OCL queries to find the origin of a bug (cf. (2) in Fig. 2). Finally, the formal underpinnings of CPNs enable *verification-based debugging* by state space exploration, which contains all possible firing sequences of a CPN (cf. (3) in Fig. 2). This allows the application of generally accepted behavioral properties, characterizing the nature of a certain CPN, e.g., to verify if a certain target model can be created with the given transformation logic. Whereas previous work [22–24] focused on distinct aspects of the TN formalism, this work aims at providing a comprehensive consideration of the diverse debugging features.

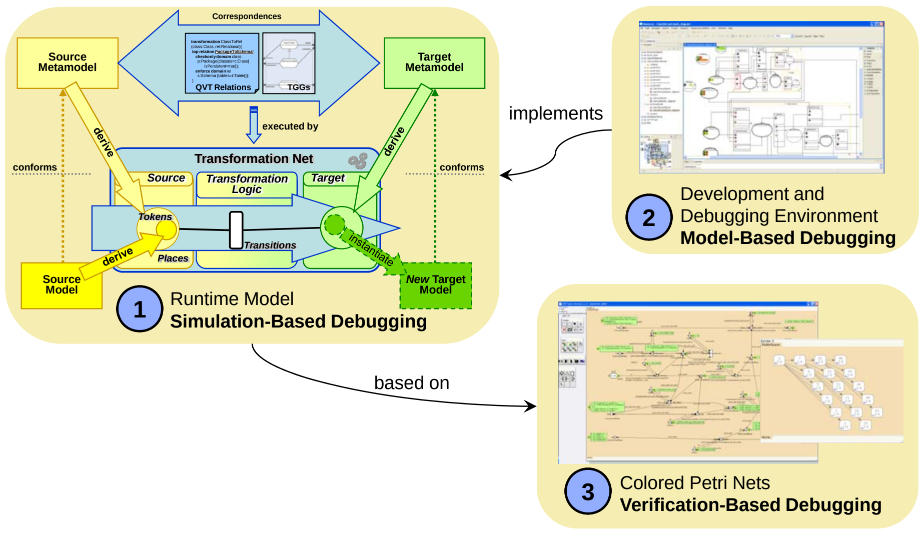


Fig. 2. Expected Main Contributions

The rest of the paper is structured as follows. Section 2 presents the concepts of the runtime model. Subsequently, Section 3 shows how to exploit the runtime model for model-based debugging. In Section 4, we introduce properties of CPNs, which can be used to formally verify model transformations. Section 5 reports on lessons learned, whereas Section 6 discusses related work. Finally, Section 7 provides an outlook on future work.

2 Runtime Model for Simulation-Based Debugging

In this section, we introduce the proposed runtime model by first describing the static parts thereof, i.e., metamodels and models, followed by the dynamic parts, i.e., transformation logic itself. In order to exemplify this, Fig. 3 depicts a small excerpt of the metamodels of the well-known UML2RDBMS case study [17] serving as a simple running example throughout the rest of the paper.

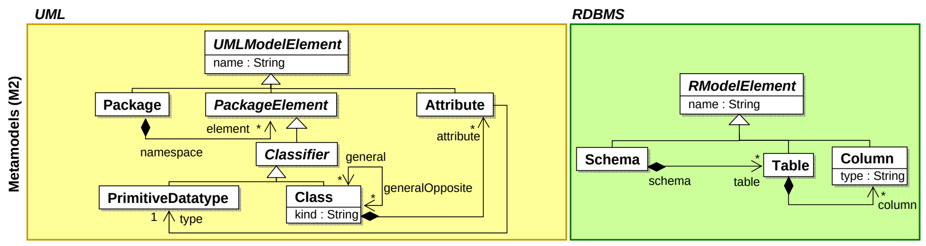
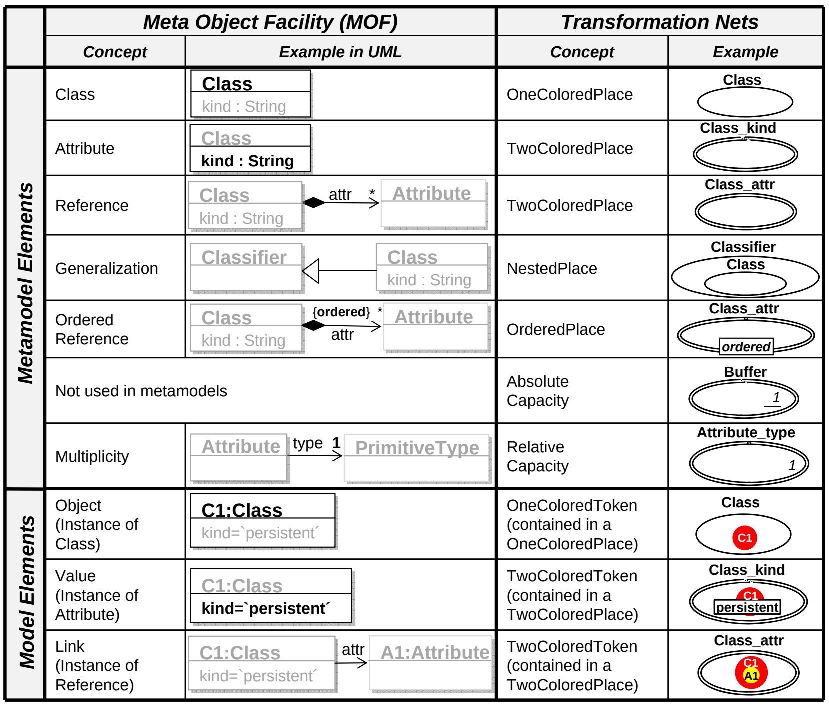


Fig. 3. Metamodels of the UML2RDBMS Example

Static Parts. When employing TNs, the static parts of a model transformation, i.e., metamodels and models, need to be represented in our formalism. This incurs transitioning from the graph-based paradigm underlying MDE into the set-based paradigm underlying Petri Nets (cf. Fig. 4). The abstract syntax of TNs is formalized by means of a metamodel (see [24]) conforming to the Ecore meta-metamodel, the Eclipse realization of OMG's MOF standard. The design rationale behind this transition is the following: We rely on the core concepts of an object-oriented meta-metamodel (MOF), 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, data values and links. Therefore, we distinguish between *one-colored places* containing *one-colored tokens* for representing the nodes of graphs, i.e., objects, and *two-colored places* containing *two-colored tokens*. These two-colored tokens represent on the one hand links between 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 one the actual value. The color is realized through a unique value that is derived from the object's id.

TNs also support modeling concepts beyond classes, attributes and references. Subclass relationships are currently represented by *nested places* whereby the place corresponding to the subclass is contained within the place corresponding to the superclass. The tokens contained in the subplace may also act as input tokens for a transition connected to the superplace. For deep inheritance hierarchies we represent the classes by separated places whereby the is-a relationship between objects is expressed by duplicating the tokens in every place involved in the inheritance hierarchy, e.g., for representing a class object, a token is put into the place `Class` and another, same-colored, token into the place `Classifier`. Ordered references in the metamodel are represented similar to ordinary references but induce a FIFO semantics when firing the transition in the TN. To represent multiplicities of references defined in the metamodel accordingly, places may be annotated with a *relative capacity* constraint to restrict the number of tokens of a specific outer-color. Finally, places may be annotated with an *absolute capacity* to restrict the total number of their tokens, e.g., for enforcing sequential processing.



	Meta Object Facility (MOF)		Transformation Nets	
	Concept	Example in UML	Concept	Example
Metamodel Elements	Class	Class / kind : String	OneColoredPlace	Class
Metamodel Elements	Attribute	Class / kind : String	TwoColoredPlace	Class_kind
Metamodel Elements	Reference	Class / kind : String; attr *; Attribute	TwoColoredPlace	Class_attr
Metamodel Elements	Generalization	Classifier; Class / kind : String	NestedPlace	Classifier; Class
Metamodel Elements	Ordered Reference	Class / kind : String; {ordered} *; attr; Attribute	OrderedPlace	Class_attr; ordered
Metamodel Elements	Not used in metamodels		Absolute Capacity	Buffer; 1
Metamodel Elements	Multiplicity	Attribute; type 1; PrimitiveType	Relative Capacity	Attribute_type; 1
Model Elements	Object (Instance of Class)	C1:Class / kind=\`persistent´	OneColoredToken (contained in a OneColoredPlace)	Class; C1
Model Elements	Value (Instance of Attribute)	C1:Class / kind=\`persistent´	TwoColoredToken (contained in a TwoColoredPlace)	Class_kind; C1; persistent
Model Elements	Link (Instance of Reference)	C1:Class / kind=\`persistent´; attr; A1:Attribute	TwoColoredToken (contained in a TwoColoredPlace)	Class_attr; C1; A1

Fig. 4. Representing MOF concepts within Transformation Nets

Dynamic Parts. The transformation logic is embodied by Petri Net transitions and additional trace places which reside in-between those places representing the original input and output metamodels. A transition consists of input placeholders (LHS of the transition) representing its pre-condition, whereas output placeholders (RHS of the transition) depict its post-condition. To express these conditions, *meta tokens* are used, prescribing a certain token configuration by means of color patterns. By matching a certain token configuration from the input places, i.e., fulfilling the pre-condition, the transition is ready to fire. The production of output tokens fulfilling the post-condition once a transition fires depends on the matched input tokens. Finally, TNs exhibit a specific firing behavior where tokens are not consumed per default since typically all combinations of tokens fulfilling a certain precondition are desired in a transformation scenario, e.g., to correctly resolve 1:n relationships between objects. This is realized by a transition's history which logs the already fired token combinations.

Since meta tokens are just typed to one-colored and two-colored tokens, but not to certain metamodel classes, transitions can form reusable transformation patterns which can be applied in different scenarios (cf. Fig. 5). For example, when a simple one-to-one correspondence should be implemented, the colors of input and output meta tokens are equal, meaning that a token is streamed through the transition only, e.g., to map `Attributes` to `Columns` (cf. *Streamer* in Fig. 5(a)). In order to set inverted references, transition (b) in Fig. 5 matches a two-colored token from its input place, and produces an inverted token in the output place, thus building up the *Inverter* pattern. 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, transition (c) matches two-colored tokens from input places and peels off the outer color of the token (cf. *Peeler* in Fig. 5). Finally, transition (d) represents a variation of the Streamer pattern called *ConditionalStreamer* adding additional preconditions. For example, this pattern may be used to ensure that objects have been created before a link can be set between them. These patterns represent basic building blocks recurring in diverse model transformation scenarios. Complementary research focuses on how these fine grained patterns can be employed in more coarse grained reusable mapping operators [14].

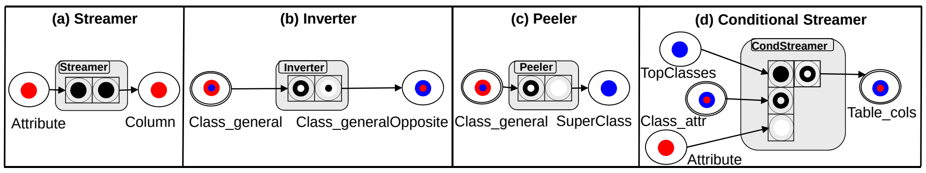


Fig. 5. Transitions forming Transformation Patterns

Summarizing, TNs allow the transformation process to be executed stepwise revealing which tokens enable a certain transition and which tokens get produced by firing this transition, enabling *simulation-based debugging*. This is possible because TNs provide a white-box view on model transformation execution, i.e., the specification needs not to be translated into another low-level executable artifact but can be executed right away. As already mentioned, this runtime might act as an execution engine for various declarative transformation languages which have a close correspondence to TNs, e.g., for QVT Relations as shown in [23], to benefit from our debugging features.

3 Development Environment for Model-Based Debugging

For exemplifying model-based debugging features integrated into a development environment, we make use of the running example, whereby an OR-mapping strategy of creating a table for every persistent class is pursued. Moreover, attributes inherited from base classes should result in additional columns. As shown in Fig. 6, our example input model comprises three classes whereby class `C2` inherits from class `C1` and class `C3` inherits from class `C2` (cf. link `general`) but only class `C1` and `C2` are marked persistent. Therefore, the desired output model should contain two `Tables`, one for class `C1` and one for class `C2`, whereby table

C2 gets an additional column originating from the attribute `name` of class C1. In the following we use this example to demonstrate our debugging facilities.

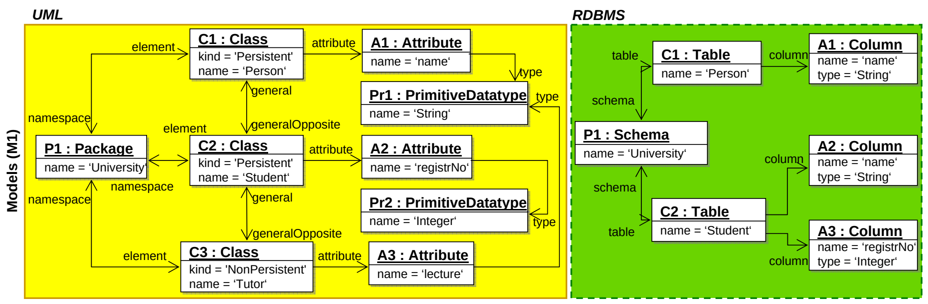
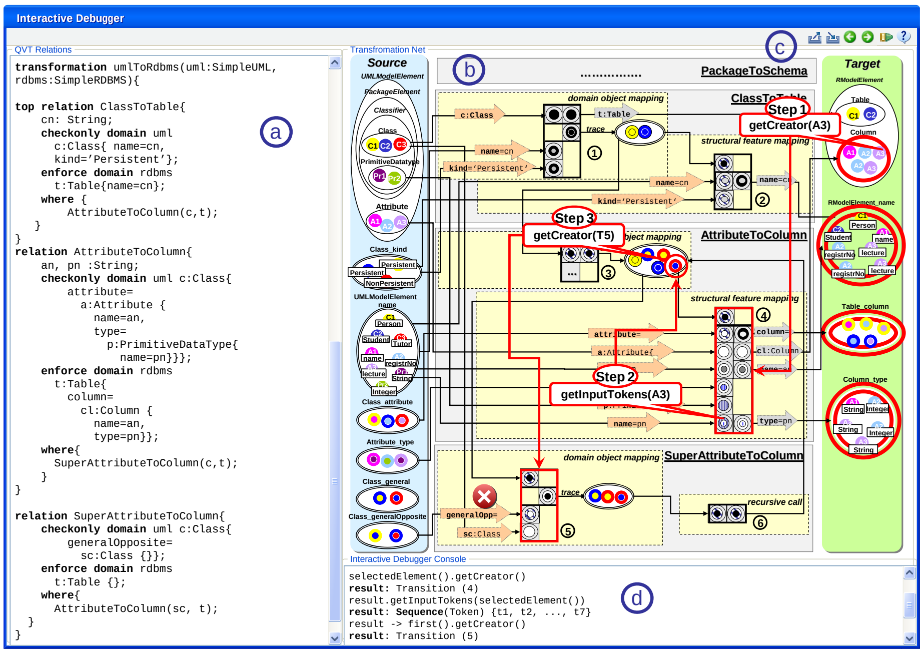


Fig. 6. Source and Target Models – Transforming Persistent Classes only.

Our development and debugging environment is based on Eclipse including two editors, one presenting the transformation specification, e.g, QVT Relations code (cf. Fig. 7(a)) and another one that shows the graphical representation thereof in terms of TNs (cf. Fig. 7(b)), realizing the above example. To provide common debugging functionalities, such as stepwise debugging, an editor toolbar (cf. Fig. 7(c)) is offered. Furthermore, functionalities to save the generated target model, i.e., to switch from the token representation to a model representation, and to load a new source model into the debugging environment are included.

Besides these standard functionalities, there are additional debugging features resulting as a benefit of using a dedicated runtime model (cf. *model-based debugging*). In particular, OCL is employed for two different debugging purposes. First, OCL is used to define conditional breakpoints at different levels of granularity. Thus, it cannot only be defined that execution should stop if a certain token is streamed into a certain place, but also if a certain combination of tokens occurs. Second, OCL is used to tackle the well-known problem in debugging that programs execute forwards in time whereas programmers must reason backwards in time to find the origin of a bug. For this, a dedicated debugging console based on the Interactive OCL Console of Eclipse (cf. Fig. 7(d)) is supported, providing several pre-defined debugging functions to explore and to understand the history of a transformation by determining and tracking paths of produced tokens [23].

The TN in its final state realizing the introduced example is depicted in Fig. 7(b). By comparing this generated target model to the expected one (cf. Fig. 6) we can see that more columns have been created than intended, e.g., two tokens labeled with `A3` in the place `Column`. The graphical representation shows that the tokens have been created by transition (4) but we do not know which source tokens were used to create exactly these target tokens. To get this information, the transformation designer may use the interactive debugging console. Within this console, s/he can use standard OCL functions and pre-defined OCL debugging functions to formulate queries that can be invoked on the runtime model.



```
transformation umlToRdbms(uml:SimpleUML,
rdbms:SimpleRDBMS){

top relation ClassToTable{
    cn: String;
    checkonly domain uml
      c:Class{ name=cn,
      kind='Persistent'};
    enforce domain rdbms
      t:Table{name=cn};
    where {
        AttributeToColumn(c,t);
   }
}
relation AttributeToColumn{
    an, pn :String;
    checkonly domain uml c:Class{
        attribute=
          a:Attribute {
            name=an,
            type=
              p:PrimitiveDataType{
                name=pn}}};
    enforce domain rdbms
      t:Table{
        column=
          cl:Column {
            name=an,
            type=pn}};
    where{
      SuperAttributeToColumn(c,t);
    }
}

relation SuperAttributeToColumn{
    checkonly domain uml c:Class{
        generalOpposite=
          sc:Class {}};
    enforce domain rdbms
      t:Table {};
    where{
      AttributeToColumn(sc, t);
  }
}
```

```
selectedElement().getCreator()
result: Transition (4)
result.getInputTokens(selectedElement())
result: Sequence(Token) {t1, t2, ..., t7}
result -> first().getCreator()
result: Transition (5)
```

Fig. 7. Development Environment for Transformation Nets.

In our example, the transformation designer selects the top-right token labeled with `A3` of the place `Column`, representing a wrongly created column `lecture` and invokes the function `getCreator(A3)` (cf. Step 1 in Fig. 7) which highlights transition (4) in the editor and additionally returns the result of the function in the debugging console. This transition reads an input token from a trace place (first LHS meta token) which receives tokens from transition (3) and (6). Therefore, it is unclear which one of these two transitions is responsible for providing the trace token used for creating the selected `Column` token.

To determine the responsible transition, the developer invokes the function `getInputTokens(A3)` on transition (4) (cf. Step 2 in Fig. 7) returning a sequence of input tokens which has been bound to produce the token `A3`. The elements in this sequence are ordered with respect to their graphical location within the transition, thus the first token in the sequence is the token that matched the top-most LHS metatoken. To get this token, the transformation designer applies the standard OCL function `first()` on the previously computed sequence which returns the single trace token. Now, the transformation designer applies the function `getCreator(T5)` (cf. Step 3 in Fig. 7) to determine which transition is responsible for producing this trace token, which yields transition (5), as transition (6) only streams the token through. Taking a closer look at transition (5), one can see that this transition uses tokens of the wrong source place, namely `Class_generalOpposite` instead of `Class_general` (see error sign in Fig. 7(b)), being the origin of the problem. The respective QVT code selects the super class instead of the base class causing the inclusion of wrong columns to the table.

4 Formal Properties for Verification-Based Debugging

Because TNs are based on CPNs, formal properties for verifying the correctness of model transformations can be applied (cf. *verification-based debugging*). For this, the state space of the CPN, being a directed graph with a node for each reachable marking and an arc for each occurring binding element, has to be constructed to calculate diverse behavioral properties.

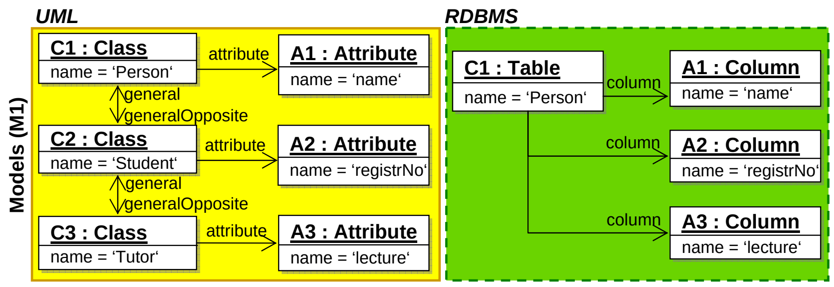
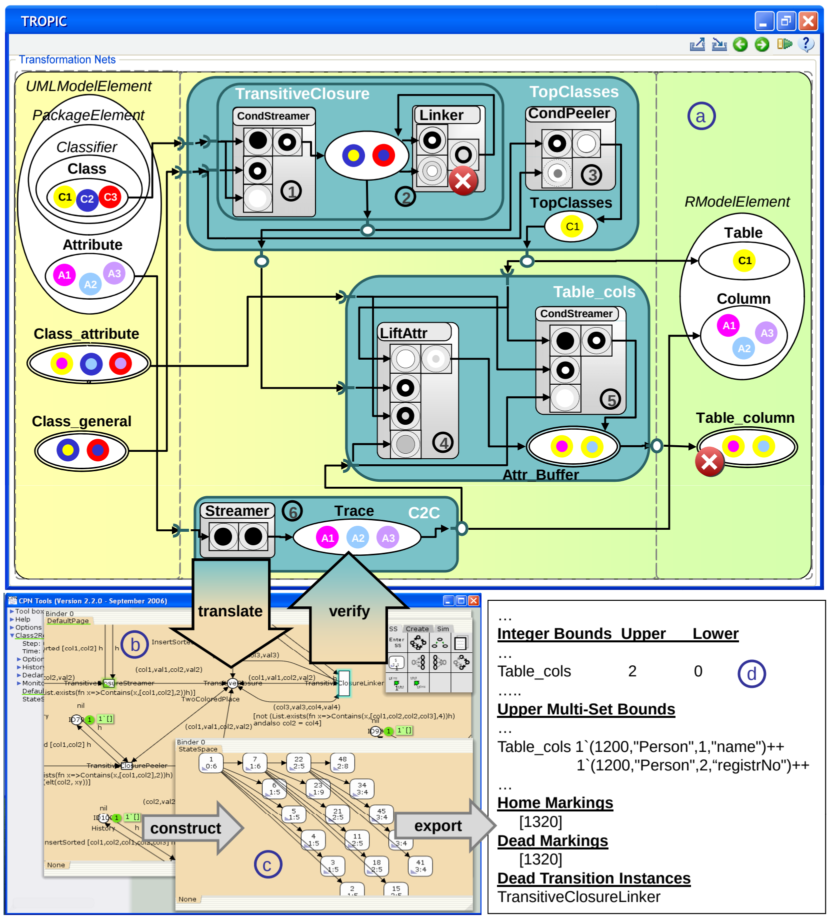


Fig. 8. Source and Target Models – Transforming Top Classes only

In order to exemplify the verification potential, we make again use of the running example – this time pursuing a different OR-mapping strategy, namely a one-table-per-hierarchy approach according to the case study presented in [2]. As shown in Fig. 8, our example comprises the same input classes and attributes as introduced before. Therefore, the desired output model should now contain one `Table`, aggregating three `Columns` (all attributes of the three classes). At a first glance the generated target model in Fig. 9(a) seems to be correct, but a closer look reveals that a link from table `C1` to column `A3` is missing (cf. missing two-colored token in place `Table_column`), compared to the desired target model depicted in Fig. 8. Even in this small example the error is hard to observe manually, suggesting the need for formal verification. In order to accomplish this, the specified transformation logic in TNs is translated to a corresponding CPN (see Fig. 9(b)), which allows to reuse formal verification capabilities of existing Petri Net engines, e.g., CPN Tools[3]. The first step in the verification process is the calculation of the state space (see Fig. 9(c)), further on used to determine behavioral properties, i.e. the verification process is based on a certain input model (see Fig. 9(d)). In the following, we show how these properties (cf. [16] for an overview) can be used to enable verification-based debugging.

Model Comparison using Boundedness Properties. Typically, the first step in verifying the correctness of a transformation specification is to compare the target model generated by the transformation to the expected target model. To identify wrong or missing target elements in terms of tokens automatically, *Boundedness* properties (*Integer Bounds* and *Multiset Bounds*) can be applied. In our example, the upper Integer Bound of the `Table_cols` place is two (cf. Fig. 9(d)) whereas the desired target model requires three tokens, as every column has to belong to a certain table. By inspecting the Multiset Bounds one recognizes

[3] http://wiki.daimi.au.dk/cpntools/cpntools.wiki



```
...
Integer Bounds  Upper     Lower
...
Table_cols          2          0
.....
Upper Multi-Set Bounds
...
Table_cols 1`(1200,"Person",1,"name")++
           1`(1200,"Person",2,"registrNo")++
...
Home Markings
     [1320]
Dead Markings
     [1320]
Dead Transition Instances
TransitiveClosureLinker
```

Fig. 9. Transformation Verification Prototype showing the UML2RDBMS example

that a link to the column A3 originating from an attribute of class C3 is missing. If such erroneous parts of the target model are detected, the owning target place is highlighted in the TN (see error sign besides the `Table_cols` place in Fig. 9(a)). Properties going beyond concrete input models can be checked by using custom functions, e.g., expressing that for all possible corresponding input and output models the number of attributes in the input model is always equal to the number of columns in the output model.

Transition Error Detection using Liveness Properties. Errors in the transformation specification occur if either a transition is specified incorrectly or the source model is incorrect. Both cases might lead to transitions which are never enabled during execution, so called *Dead Transition Instances* or *L0-Liveness* [16]. The state space report in Fig. 9(d) shows that transition 2 in the TN is a *Dead Transition Instance*, which is therefore marked with an error sign. The intention of transition 2 in our example is to calculate the transitive closure, thus there should be an additional link from class C3 to class C1 as class C3 also inherits from class C1 (see Fig. 8). When investigating the LHS of transition 2 in Fig. 9(a) we see that the inheritance hierarchy is faulty; the pattern specifies that a class X (white color) is parent of a class Y (black color) and of a class Z (gray color). To fix the color pattern we need to change the outer and inner colors of the second input token of the transition. After fixing the error, the state space can be constructed again and will not contain any dead transitions anymore.

Termination and Confluence Verification using Dead and Home Markings. A transformation specification must always be able to terminate, thus the state space has to contain at least one *Dead Marking*. This is typically ensured by the history concept of TNs, which prevents firing recurring token combinations. 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* property requiring that a marking M can be reached from any other reachable marking.

The generated report in Fig. 9(d) shows that in our example a single *Home Marking* is available which is equal to the single *Dead Marking* (both identified by the marking 1320), meaning that the transformation specification always terminates and is confluent. To achieve a correct transformation result, an equal *Home Marking* and *Dead Marking* is a necessary but not a sufficient condition, as it cannot be ensured that this marking represents the desired target model. By exploring the constructed state space, it is possible to detect if a certain marking, i.e., the target marking derived from the desired target model, is reachable with the specified transformation logic. If this is the case, and if this marking is equal to both, *Home Marking* and *Dead Marking*, it is ensured that the desired target model is created with the specified transformation logic in any case.

5 Lessons Learned

This section presents lessons learned from already conducted transformation examples with our TN formalism and thereby discusses key features as well as current limitations of the TN approach.

Colored Tokens Representing Model Elements Reveal Traceability. The source model to be transformed is represented by means of one-colored tokens and two-colored tokens residing in the source places of the TN whereby the actual transformation is performed by streaming these tokens to the target places. Through this mechanism traceability is ensured since the source – target relationship can be derived by simply searching for same-colored tokens in source places and target places, respectively.

Visual Syntax and Live Programming Fosters Debugging. TNs provide a visual formalism for defining model transformations which is especially useful for debugging purposes. Since the flow of model elements undergoing certain transformations can be directly followed by observing the flow of tokens, undesired results can be detected easily. Another characteristic of TNs that facilitates debugging 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 TN by just setting up a suitable token configuration in the input places.

Implicit Control Flow Eases Evolution. The control flow in a TN 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 form of rules) synchronized. For instance, when a rule needs to be changed to match for additional model elements, one would have to take care to call this rule at a time when the elements to be matched already exist.

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 color patterns of the meta tokens of the input placeholders and the output placeholders. On the other hand, the larger the TN 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.

State Space Explosion Limits Model Size. A known problem of formal verification by 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, Stubborn Sets [13] could be applied to reduce the state space nearly to half size, thus enhancing scalability of our approach.

6 Related Work

In the following, related work regarding (i) debugging support, (ii) understandability, and (iii) formal verification of transformation languages is discussed.

Debugging Support of Transformation Languages. In general, there is little debugging support for transformation languages. 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 [21] 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 [7], 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. In addition to that, Fujaba supports visualization of how the graph evolves during transformation, and allows interactive application of transformation rules. Approaches like VIATRA [1], which produce debug reports that trace an execution, only, but do not provide interactive debugging facilities. Although the ATL debugger [11] allows the step-wise execution of ATL byte-code, only low-level debugging information is provided, e.g., variable values. This limited view hinders observing the execution of the whole transformation, e.g., the coherence between different rules. SmartQVT and TefKat [15] allow for similar debugging functionality.

Hibberd et al. [8] present forensic debugging techniques for model transformations by utilizing the trace information of model transformation executions for determining the relationships between source elements, target elements, and the involved transformation logic. With the help of such trace information, it is possible to answer debugging questions implemented as queries which are important for localizing bugs. In addition, they present a technique based on program slicing for further narrowing the area where a bug might be located. The work of Hibberd et al. is orthogonal to our approach, because we are using live debugging techniques instead of forensic mechanisms. However, our approach allows to answer debugging questions based on the visualization of the path a source token has taken to become a target token.

Summarizing, what sets TNs apart from these approaches is that all debugging activities are carried out on a single integrated formalism, without the need to deal with several different views. Although there are declarative approaches based on relational languages, e.g., Prolog-based approaches [18], that do not induce a gap between the specification and the execution, their debugging features are considered not that adequate for our purposes since the unification and backtracking processes in Prolog give rise to the possibility of an increased confusion about the location of errors [4]. Furthermore, our 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 TN itself. This ability for live-programming enables an additional benefit for debugging and development: one can correct errors (e.g., stuck tokens) in TNs right away without needing to recompile and restart the debug cycle.

Understandability of Transformation Languages. 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.

Verification Support of Transformation Languages. Especially in the area of graph transformations, some work has been conducted using Petri Nets to check formal properties of graph transformation rules. Thereby, the approach proposed in [20] translates individual graph transformation 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 models of the production system, as well as the graph transformation rules are transformed into Petri Nets in order to make use of the formal verification techniques for checking properties of the production system models. Varró [19] presents a translation of graph transformation rules to transition systems (TS), serving as the mathematical formalism of various different model checkers to achieve formal verification of model transformations. Thereby, only the dynamic parts of the graph transformation systems are transformed to TS to reduce the state space.

Summarizing, these approaches only check for confluence and termination of the specified graph transformation rules, but compared to our approach, they do not consider additional properties which might be helpful to point out the origin of an error. Additionally, these approaches are using Petri Nets only as a back-end for automatically analyzing properties of transformations, whereas we are using TNs as a front-end for fostering debuggability.

7 Future Work

Since the research is in its initial phase, currently a first prototype is available only, which should be evaluated on basis of the following two research questions:

Question 1: *Does the proposed approach foster the detection of bugs?* Regarding this issue, an empirical study will be conducted with students from our model engineering courses (around 200 master students). The aim of this empirical study is to evaluate if the offered debugging facilities lead to decreased time required for detecting a certain bug. In this respect, the students will be divided into subgroups and each subgroup will be provided with an erroneous example in a certain transformation language like ATL, QVT and the TN formalism. Then the students will have to debug the examples with the provided debugging facilities. The evaluation will take place by measuring the time required to find the bug. A source to acquire complex transformations scenarios for evaluating this research question could be the GraBaTs tool contest[4].

Question 2: *Do the offered verification possibilities help in finding bugs?* Concerning this question, a test set of erroneous transformation examples will be collected by asking the students of our model engineering course to not only submit the correct version of the assigned transformation example but also to submit preceding erroneous examples. Thereby, we will collect a set of erroneous examples with "real-world" bugs. This test set will then be used to check whether our proposed verification possibilities are useful in detecting these bugs.

References

1. A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *Proc. of SAC '06*, 2006.
2. J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model Transformations in Practice Workshop of MoDELS'05, Montego Bay, Jamaica, 2005.
3. E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *Proc. of MoDELS'08*, pages 53–67, 2008.
4. P. Brna, M. Brayshaw, M. Esom-Cook, P. Fung, A. Bundy, and T. Dodd. An Overview of Prolog Debugging Tools. *Instructional Science*, 20(2):193–214, 1991.

[4] http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/

5. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
6. J. de Lara and H. Vangheluwe. Translating Model Simulators to Analysis Models. In *Proc. of 11th Int. Conf. on Fundamental Approaches to Software Engineering*, pages 77–92, Budapest, Hungary, April 2008.
7. L. Geiger. Model Level Debugging with Fujaba. In *Proc. of 6th Int. Fujaba Days*, pages 23–28, Dresden, Germany, September 2008.
8. M. T. Hibberd, M. J. Lawley, and K. Raymond. Forensic Debugging of Model Transformations. In *Proc. of MoDELS'07*, pages 589–604, Nashville, USA, 2007.
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. F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proc. of the Model Transformations in Practice Workshop at MoDELS'05*, pages 128–138, Montego Bay, Jamaica, 2005.
12. A. Koenigs. Model Transformation with TGGs. In *Proc. of Model Transformations in Practice Workshop of MoDELS'05*, Montego Bay, Jamaica, 2005.
13. L. Kristensen and A. Valmari. Finding Stubborn Sets of Coloured Petri Nets without Unfolding. In *Proc. of Int. Conf. on Application and Theory of Petri Nets*, pages 104–123. London, UK, 1998.
14. A. Kusel. TROPIC - A Framework for Building Reusable Transformation Components. Proc. of the Doctoral Symposium at MoDELS, Technical Report 2009-566, School of Computing, Queen's University, Kingston, Canada, 2009.
15. M. Lawley and J. Steel. Practical Declarative Model Transformation with Tefkat. *Model Transformations in Practice Workshop of MoDELS'05*, pages 139–150, 2005.
16. T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4), 1989.
17. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. www.omg.org/docs/ptc/07-07-07.pdf, 2007.
18. B. Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. In *Proc. of SLE'08*, 2008.
19. D. Varró. Automated Formal Verification of Visual Modeling Languages by Model Checking. *Journal of Software and Systems Modelling*, 3(2):85–113, 2003.
20. D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformation by Petri Nets. In *Proc. of Int. Conf. on Graph Transformation*, pages 260–274, Natal, Brazil, 2006.
21. R. Wagner. Developing Model Transformations with Fujaba. In *Proc. of the 4th Int. Fujaba Days 2006*, pages 79–82, Bayreuth, Germany, 2006.
22. 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 OOPSLA Workshop on Domain-Specific Modeling*, Orlando, USA, 2009.
23. M. Wimmer, G. Kappel, J. Schönböck, A. Kusel, W. Retschitzegger, and W. Schwinger. A Petri Net based Debugging Environment for QVT Relations. In *Proc. of the 24th Int. Conf. on ASE'09*, pages 1–12, 2009.
24. M. Wimmer, A. Kusel, T. Reiter, W. Retschitzegger, W. Schwinger, and G. Kappel. Lost in Translation? Transformation Nets to the Rescue! In *Proc. of 3rd Int. United Information Systems Conf.*, pages 315–327, Sydney, Australia, 2009.