Right or Wrong? – Verification of Model Transformations using Colored Petri Nets*

M. Wimmer
TU Vienna
wimmer@big.tuwien.ac.at

G. Kappel
TU Vienna
gerti@big.tuwien.ac.at

A. Kusel
JKU Linz
kusel@bioinf.jku.at

W. Retschitzegger
University of Vienna
werner@bioinf.jku.at

J. Schoenboeck
TU Vienna
schoenboeck@bioinf.jku.at

W. Schwinger
JKU Linz
wieland@schwinger.at

ABSTRACT

Model-Driven Engineering (MDE) places models as first-class artifacts throughout the software lifecycle requiring the availability of proper transformation languages. Most of today's approaches use declarative rules to specify a mapping between source and target models which is then executed by a transformation engine. Transformation engines, however, most often hide the operational semantics of the mapping and operate on a considerable lower level of abstraction, thus hampering debugging. To tackle these limitations we propose a framework called TROPIC (Transformations on Petri Nets in Color) providing a DSL on top of Colored Petri Nets (CPNs) to specify, simulate, and formally verify model transformations. The formal underpinnings of CPNs enables simulation and verification of model transformations. By exploring the constructed state space of CPNs we show how predefined behavioral properties as well as custom state space functions can be applied for observing and tracking origins of errors during debugging.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Model Transformation, CPN, Verification, Debugging

1. INTRODUCTION

MDE places models as first-class artifacts throughout the software life cycle, whereby model transformation languages play a vital role [5]. Several kinds of dedicated transformation languages are available (see [2] for an overview), the majority of them favoring declarative, rule based specifications to express mappings between source and target models, as is the case with the QVT Relations standard [1].

To execute the specified mapping, transformation engines are used, hiding the operational semantics and operating on a considerable lower level of abstraction. On the one hand this relieves transformation designers from burdens, like the necessity to specify a certain execution order. On the other hand, as transformation specifications grow larger, requiring numerous rules working together, this considerably hampers observing, tracking origins, and fixing of possible errors, being the main phases of debugging. As the correctness of the automatically generated target model fully depends on the correctness of the specified model transformation [13], formal underpinnings are required to enable verification of model transformations by proving certain properties like confluence and termination, to ease debugging [11].

To alleviate the above mentioned problems we propose TROPIC (TRansformations On Petri Nets In Color) [17, 18, 19], a framework providing declarative, reusable mapping operators based on a DSL on top of Colored Petri Nets (CPNs) [6] called Transformation Nets (TNs) to specify model transformations. TNs provide a homogenous representation of declarative mapping operators and their operational semantics, both in terms of CPN concepts. The formal underpinning of CPNs enables simulation of model transformations and exploration of the state space, which shows all possible firing sequences of a CPN. This allows applying generally accepted behavioral properties, characterizing the nature of a certain CPN, e.g., with respect to confluence or termination, as well as custom functions, e.g., to check if a certain target model can be created with the given transformation logic, during the observation and tracking origin phase of debugging.

The remainder of this paper is structured as follows. Section 2 introduces the main concepts of TROPIC. In Section 3, we show how properties of CPNs can be used to formally verify model transformations. Section 4 provides a taxonomy of possible transformation errors and related CPN properties, whereas Section 5 reports on lessons learned. Related work is discussed in Section 6, and finally, Section 7 provides an outlook on future work.

2. TROPIC IN A NUTSHELL

In the following, we shortly introduce TROPIC, a framework for model transformations applying CPN concepts. The use of Petri Nets in general enables a process oriented

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

DSM Workshop at OOPSLA '09 Orlando, Florida USA

view on transformations. The abstraction from control flow prevalent in declarative transformation approaches is achieved as transitions can fire autonomously depending on the markings contained in the places only, although the statefulness of imperative approaches is preserved. CPNs, being a well-known class of Petri Nets, are most suited for model transformations, since every token carries a value of a certain type called token color, used to represent model elements accordingly. Thus CPNs provide a runtime model allowing transformation designers to gain an explicit, integrated representation of the operational semantics of model transformations, which particularly favors debugging. To profit from these benefits of CPNs while hiding low-level details and circumventing restrictions thereof with respect to model transformations (cf. below), the TROPIC framework introduces a DSL for model transformations called Transformation Net (TN). TNs operate on two different levels of abstraction, providing a high-level mapping view and a more detailed transformation view (see Fig. 1).

Mapping View. The mapping view (upper part of Fig. 1) is used to declaratively define the correspondences between source (LHS) and target metamodel elements (RHS) using mapping operators (MOPs) encapsulating recurring transformation logic. MOPs are represented by means of *Hiearchical CPN concepts* [6], providing a packaging mechanism allowing a black-box view hiding the operational semantics of a transformation and a white-box view making the operational semantics explicitly (cf. below). In addition, a MOP's interface is only typed by classes, attributes and references being the main constituents of the Ecore meta-metamodel. This "weak typing" mechanism based on Ecore concepts allows to abstract from concrete metamodels, thus enabling reuse. For example in Fig. 1, showing a very simple transformation of classes to relations, the `C2C` MOP is used to simply map a class of the source model (`Class`) to a class in the target model (`Table`).

Transformation View. Every MOP of the mapping view requires a well defined operational semantics (i.e., the white-box view) in the form of some executable piece of transformation logic realized by an independent set of transitions and places. In particular, places are derived from elements of metamodels, creating a place for every class, attribute and reference thereof (see middle part of Fig. 1). Tokens are created from elements of models and then put into the according places. Finally, transitions represent the actual transformation logic. The existence of certain model elements (i.e., tokens) allows transitions to fire and thus stream these tokens from source places to target places to set up an unidirectional transformation. Tokens in target places finally represent instances of the target metamodel to be created and additional trace information is hold in terms of tokens within trace places. Note that the tokens in the target and trace places in Fig. 1 represent a successfully executed transformation. The LHS of a transtion representing the pre-conditions as well as the RHS depicting the post-conditions are visualized by means of color patterns (called `MetaTokens`). If a transition is enabled, the colors of the input tokens are bound to the input pattern. The production of output tokens is typically dependent on the matched input tokens. For instance, if a transition simply streams a certain token indicated by the same color pattern of `MetaTokens` on LHS and RHS, exact the same token is produced as output that was matched at the LHS. For example, the

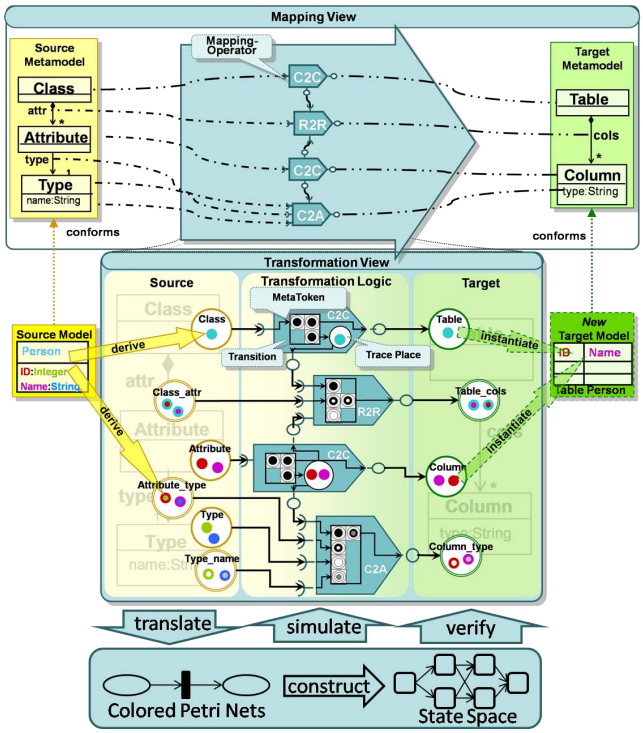

Figure 1: Conceptual Architecture of TROPIC.

`C2C` operator is realized by such a transition taking a class (cf. arc from the place `Class` to LHS `MetaToken`) and producing an according table (cf. arc from RHS `MetaToken` to place `Table`) and additional trace information. For further details on TNs we refer to [17].

DSL on top of CPNs. TNs can be fully translated into existing CPN concepts, although, as already mentioned, adaptions have been made in order to better suit the domain of model transformations. These adaptions comprise, most importantly, a specific consumption behavior in order to deal with 1:n relationships and means to represent certain modeling concepts like multiplicity, ordered references and inheritance, accordingly.

First of all, the necessity of a specific consumption behavior is motivated in Fig. 2 by means of an example, depicting a simple TN generating a table for every class contained in a package. If token `P1` would be consumed (which is the default in CPNs), the transition could only fire once and the 1:n relationship between package and class would not be correctly resolved. Therefore, TNs provide a specific consumption behavior where tokens are not consumed per default. Rather the combinations of tokens fulfilling the precondition are held as trace information by every transition which allows firing for all possible combinations, which is typically desired in transformation scenarios. To represent the consumption behavior in CPNs we use an additional `History` place storing the already fired combinations in a sorted list together with an according guard condition, ensuring that the transition only fires if the current tokens of the precondition are not already contained in the history (see History Concept in Fig. 2b).

Regarding multiplicities of references, we provide *restrictions of places* in TNs represented by so called anti-places in CPNs. E.g., if the multiplicity of a reference in a target

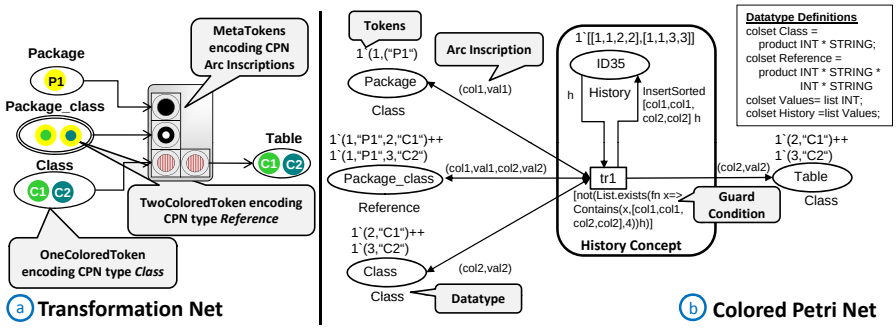

Figure 2: Translation of the Transformation Net Consumption Behavior to CPNs.

model is set to one, an anti-place holds exactly one token which is consumed if the reference is transformed, prohibiting repeated firings, cf. [9] for details. To cope with ordered references in a metamodel TNs introduce *ordered places* represented by lists in combination with anti-places in CPNs as they do not predefine a certain matching order for tokens. Inheritance between classes in a metamodel is depicted by means of *nested places* in TNs (place of superclass contains places of subclasses) and are represented by one place per class whereby places of superclasses additionally aggregate tokens of subclasses in CPNs. Additionally, to allow testing the absence of tokens, e.g., to create a class only if no link exists to a parent class (see transition c in Fig. 4a), TNs provide explicit concepts to represent *inhibitor arcs* hiding the CPN pattern presented in [9]. In contrast to metamodel specific concepts the translation of places, tokens and transitions itself is straightforward (places in TNs get converted to places with an according data type in CPNs, tokens in TNs remain tokens in CPNs, color patterns of transitions get converted to equivalent arc inscriptions, cf. Fig. 2).

3. PETRI NET-BASED VERIFICATION BY EXAMPLE

In the previous section we presented the foundations of TNs and their translation to CPNs which allows for the use of existing CPN execution engines to simulate TNs and, most importantly, the formal exploration of CPN properties. In the following subsections we present how properties of CPNs can be applied to verify model transformation specifications by means of an example.

3.1 UML2Relational Example

The example depicted in Fig. 3 and Fig. 4 is based on the Class2Relational case study[1], which became the de-facto standard example for model transformations. Due to reasons of brevity, only the most challenging part of this case study is described in this paper, namely how to represent inheritance hierarchies of classes within relational schemas following a simple one-table-per-hierarchy approach. As shown in Fig. 3 our example comprises three classes (cf. tokens in place `Class` in Fig 4a) whereby class `C2` inherits from class `C1` and class `C3` inherits from class `C2` (cf. tokens in place `Class_par` in Fig 4a). Therefore the desired output model should contain one `Table`, aggregating four `Columns` (all attributes of the three classes).

At a first glance the generated target model in Fig. 4a

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

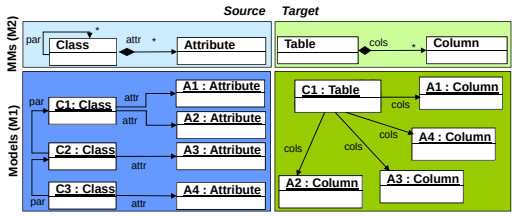

Figure 3: Metamodel and Model of UML2Relational Example.

seems to be correct, but on a closer look it can be detected, that a link from table `C1` to column `A4` is missing, compared to the desired target model depicted in Fig. 3. Even in this small example the error is hard to observe, but it is even more difficult to track the origin of the error. In the following we show how predefined formal properties of CPNs (cf. [10] for an overview) in combination with custom state space functions can ease these debugging phases using our current prototype.

3.2 Transformation Verification Prototype

As shown in Fig. 4, the created TN is translated to an according CPN, which allows the use of existing Petri Net execution engines, e.g., CPN Tools [2], enabling the simulation of model transformations. Although simulation can be used to get a first insight into the transformation specification, i.e., to investigate the operational semantics of the specified transformation, it is impossible to obtain a complete proof of *behavioral properties*, which require formal analysis methods of CPNs. Therefore the state space is constructed (see Fig. 4e), used on the hand to obtain predefined properties (see Fig. 4f), and on the other hand to analyze the transformation specification using custom functions, e.g., to check if a certain marking is reachable. The *Transformation Analyzer* component (see Fig. 4b) processes the analysis results, thereby verifying the specified transformation. Additionally to a source model and the specified transformation logic needed to calculate the state space, we assume that the expected target model is known, which is loaded by the *Transformation Analyzer* to derive the desired target markings in CPNs, which is then used for testing the transformation logic by applying custom state space functions.

3.3 Verification of Model Transformations

In the following we show, how formal properties can be applied to detect errors in the transformation specification.

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

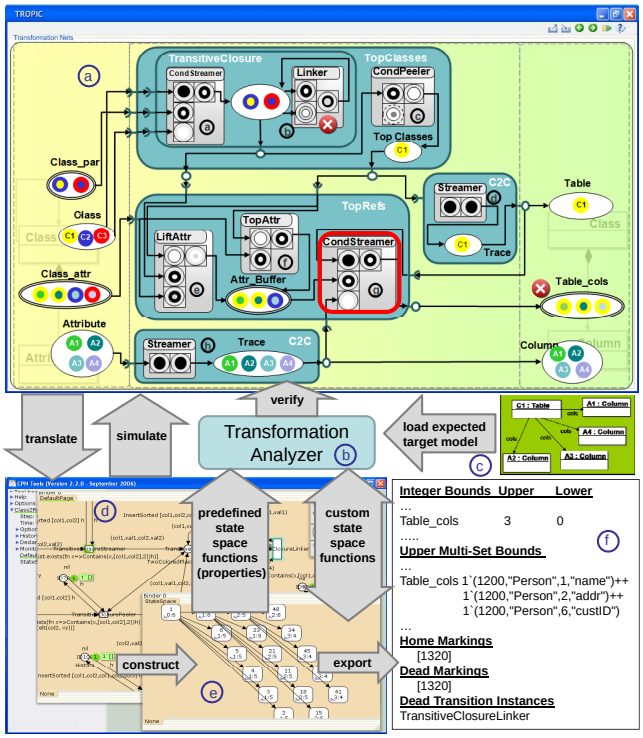

Figure 4: Transformation Verification Prototype showing the UML2Relational example

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, manually created 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 (cf. Fig. 4f), the upper integer bound of the `Table_cols` place is set to three whereas the desired target model requires four tokens, as every column has to belong to a certain table. By inspecting the multiset bounds one recognizes that a link to the column `A4` originating from an attribute of class `C3` is missing. If such erroneous parts of the target model are detected, the owning target place (see error sign besides the `Table_cols` place in Fig. 4a) as well as the transitions that produce tokens in these places are highlighted in the TN. Unfortunately, numerous transitions are involved in creating the `Table_cols` link in our example, which hampers finding the actual origin of the error.

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*. The state space report in Fig. 4f shows that transition b in the TN is a *Dead Transition Instance*, which is therefore marked with an error sign. The intention of transition b in our example is to calculate the transitive closure, thus there should be an additional link from class `C` to class `A` as class `C` also inherits from class `A` (see Fig 3). On investigating the LHS of transition b in Fig. 4 we see that the inheritance hierarchy is faulty; the pattern specifies that class `X` (white color) is parent of class `Y` (black color) and class `Z` (gray color). To fix the color pattern we need to change outer and inner color of the second Meta-Token; now class `X` (white color) is parent of class `Y` (black color), and `X` is again parent of class `Z` (gray color). After fixing the error, the state space can be constructed again and will not contain 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 for recurring combinations. Finally 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. 4f 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. 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 using custom functions it is possible to detect if a certain marking is reachable with the specified transformation logic, i.e., the target marking derived from the desired target model. If it is, and the marking is equal to both, *Home Marking* and *Dead Marking*, it is ensured that the desired target model can be created with the specified transformation logic in any case.

4. CPN PROPERTIES FOR MODEL TRANSFORMATIONS

By applying and analyzing behavioral properties of CPNs in different case studies we tried to figure out which properties are useful in the context of model transformation verification and which kind of errors can be detected. The proposed taxonomy (see Fig. 5) investigates possible locations of errors, classifies typical model transformations errors and shows which properties are useful for their detection. The taxonomy extends our taxonomy presented in [8], focusing on how common QVT pitfalls can be detected in TROPIC.

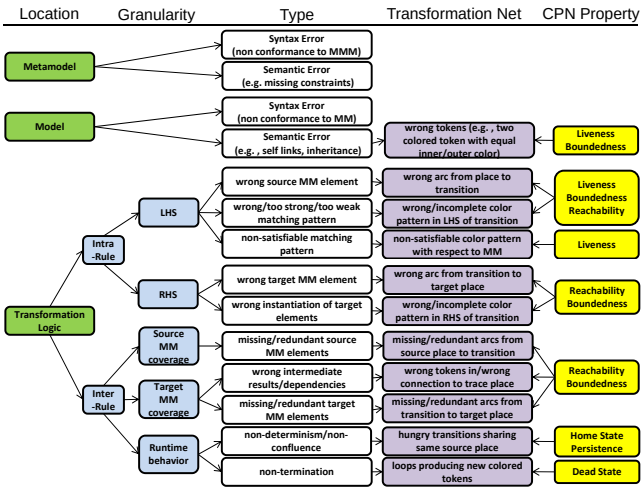

Figure 5: Taxonomy of Transformation Errors and CPN Properties.

During specification of model transformations there are three possible locations of errors, either in (i) the metamodel, (ii) the model, or (iii) the transformation logic. The detection of errors in the metamodel is in general out of scope of transformation languages. As we explicitly represent model elements—in contrast to other transformation languages—as tokens in TNs, semantic errors in the model can be detected by the liveness or boundedness property. For example, an incorrect source model (e.g., self links represented by two colored tokens with same inner and outer color) might lead to dead transition instances or an incorrect firing behavior of a transition and thus to an incorrect number of tokens in the target place.

Errors in the transformation logic itself can be divided into errors local to a single transition (*Intra-Rule Error*) or errors which can only be detected by examining the interrelations between several transitions (*Inter-Rule Error*). Intra-rule errors can be divided into errors occurring at the LHS or RHS of a transition. Common errors on both sides (e.g., a wrong matching pattern or a wrong instantiation of target models) can be detected by examining the boundedness properties in comparison to an expected target model or by custom state space functions checking if a certain marking is reachable. Due to the fact that these two properties can be applied in various scenarios we provide special tool support. If an expected target model is loaded, boundedness properties are automatically checked. Additionally, by selecting individual tokens of the desired target model (visualized in the TN editor by the *Transformation Analyzer* component), custom state space functions are created checking if the desired marking is reachable or not with the given transformation specification and the given source model. Finally, dead transition instances point out that the a given LHS specification of a transition cannot be fulfilled by the given source model.

Inter-rule errors occur if transitions depend on other, erroneous transitions or if we miss to cover the whole source or target metamodel. Although these errors can easily be detected by checking for source places that have no arc to any transition or target places which are not target of any transitions, it is also possible to apply boundedness and reachability properties to detect these kind of errors. To verify if several transitions in a model transformation specifications interact correctly, the confluence and the termination property can be applied. The creation of non-confluent transformation specifications in TNs might only occur if several transitions explicitly consume tokens from one place. If, in this case, the *Persistence* property is violated (the firing of one transition disables the firing of another one enabled before), which would lead to non-confluent model transformations, an error notification would be given to the transformation designer. As already stated and detailed in Section 5 the specific firing behavior of TNs ensures termination.

5. LESSONS LEARNED

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

History ensures termination. As mentioned above, TNs introduce a specific consumption 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 [12], TNs ensure termination already at design time.

Visual syntax and live programming fosters debugging. TNs represent a visual formalism for defining model transformations which is, in combination with the exploration of formal properties, 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, allowing to detect undesired results easily. Another characteristic of TNs, 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.

Concurreny in Petri Nets allows parallel execution of model transformations. As Petri Nets in general are especially suitable to specify concurrent operations, parallel execution of transformation logic is possible, thereby increasing efficiency of the transformation execution. In the UML2Relational example shown in Fig. 4a we can concurrently transform attributes to columns (cf. transition h) and calculate the transitive closure (cf. transition a). The properties *Home State* and *Dead Markings* can ensure confluence, even in case of parallel execution. The chosen representation of models by TNs let attributes as well as references become first-class citizens, resulting in a fine-grained decomposition of models allowing for extensive use of parallel execution.

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 [7] could be applied to reduce the state space nearly to half size, thus enhancing scalability of our approach.

6. RELATED WORK

The main objective of this paper is to provide formal verification methods for finding common transformation problems by employing CPNs. We consider two orthogonal threads of related work. First, we discuss other approaches which provide formal verification methods for model transformations. Second, we relate our proposed taxonomy to other error taxonomies in the domain of model transformations.

Formal verification 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 [15] translates individual graph rules into a Place/-Transition Net and checks for its termination. Another approach is described in [4], 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 trans-

formed into Petri Nets in order to make use of the formal verification techniques for checking properties of the production system models. Varró presents in [14] a translation of graph transformation rules to transition systems (TS), serving as the mathematical specification formalism of various different model checkers to achieve formal verification of model transformation. Thereby, only the dynamic parts of the graph transformation systems are transformed to TS in order to reduce the state space.

These mentioned approaches only check for confluence and termination of the specified graph transformation rules, but compared to our approach, make no use of additional properties which might be helpful to e.g., 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 a DSL on top of CPNs as a front-end for fostering debuggability.

Error Taxonomies. In [13], a simple error taxonomy for model transformations is presented which is then used to automatically generate test cases for model transformations. A very similar approach is presented by Darabos et. al. in [3], focusing on common errors in graph transformation languages in general and on errors in the graph pattern matching phase in particular.

Both taxonomies are, however, rather general and describe possible errors in graph transformation specifications, only. Neither suggestions are presented how the findings of the generated test cases can be mapped back to the transformation specification in order to fix possible errors nor formal validation methods are presented.

7. FUTURE WORK

Up to now, we focused on small model transformation scenarios only, not least due to the state space explosion problem. The main disadvantage of the state space algorithms included in CPN Tools is that only full occurrence graphs can be constructed. The ASCoVeCO State space Analysis Platform (ASAP) [16], however, provides a tool to perform state space analysis on CPNs which tackles these shortcomings by allowing the specification of own, complex algorithms to construct and to explore the state space. We plan to integrate the ASAP tool into our prototype for evaluating different methods for their suitability in the domain of model transformations. Additionally, as the transformation logic by means of color patterns can easily become hard to comprehend when domain patterns grow larger, we plan to employ alternative visualization techniques, e.g., object diagrams or arc inscriptions known from CPNs.

8. REFERENCES

[1] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification, 2007.

[2] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 2006.

[3] A. Darabos, A. Pataricza, and D. Varró. Towards Testing the Implementation of Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 211, April 2008.

[4] J. de Lara and H. Vangheluwe. Automating the Transformation-Based Analysis of Visual Languages. *Formal Aspects of Computing*, 21, Mai 2009.

[5] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *29th Int. Conf. on Software Engineering*, 2007.

[6] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modeling and Validation of Concurrent Systems*. Springer, 2009.

[7] L. Kristensen and A. Valmari. Finding Stubborn Sets of Coloured Petri Nets without Unfolding. In *Poc. of Int. Conf. on Application and Theory of Petri Nets*. London, 1998.

[8] A. Kusel, W. Schwinger, M. Wimmer, and W. Retschitzegger. Common Pitfalls of Using QVT Relations - Graphical Debugging as Remedy. *Int. Workshop on UML and AADL @ ICECCS'09, Potsdam*, 2009.

[9] N. A. Mulyar and W. M. P. van der Aalst. *Patterns in Colored Petri Nets*. Beta, Research School for Operations Management and Logistics, 2005.

[10] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4), 1989.

[11] F. Orjeas, E. Guerra, J. de Lara, and H. Ehrig. Correctness, completeness and termination of pattern-based model-to-model transformation. In *Proc. of 3rd Conf. on Algebra and Coalgebra in Computer Science*, Udine, 2009.

[12] D. Plump. Termination of graph rewriting is undecidable. *Fundamental Informatics*, 33(2), 1998.

[13] J. Uster, J. M. Küster, and M. A. el razik. Validation of Model Transformations - First Experiences using a White Box Approach. In *Proc. of MoDeVa'06 at MoDELS'06*, Genova, 2006.

[14] D. Varró. Automated Formal Verification of Visual Modeling Languages by Model Checking. *Journal of Software and Systems Modelling*, 3(2), 2003.

[15] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. In *Proc. of 3rd ICGT*, Natal, 2006.

[16] M. Westergaard, S. Evangelista, and L. M. Kristensen. ASAP: An Extensible Platform for State Space Analysis. In *Proc. of 30th Int. Conf. on Application and Theory of Petri Nets and Other Models of Concurrency*, Paris, 2009.

[17] 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. (UNISCON'09)*, Sydney, 2009.

[18] M. Wimmer, A. Kusel, J. Schoenboeck, G. Kappel, W. Retschitzegger, and W. Schwinger. Reviving QVT Relations: Model-based Debugging using Colored Petri Nets. In *MoDELS '09: Proceedings of the 12th international conference on Model Driven Engineering Languages and Systems*, Denver, 2009.

[19] M. Wimmer, A. Kusel, J. Schoenboeck, T. Reiter, W. Retschitzegger, and W. Schwinger. Let's Play the Token Game – Model Transformations Powered By Transformation Nets. In *Proc. of Int. Workshop on Petri Nets and Software Engineering*, Paris, 2009.