Debugging Model-to-Model Transformations

J. Schönböck, G. Kappel, and M. Wimmer
Vienna University of Technology, Austria
Email: {lastname}@big.tuwien.ac.at

A. Kusel, W. Retschitzegger,
and W. Schwinger
Johannes Kepler University Linz, Austria
Email: {firstname.lastname}@jku.at

***Abstract*—Model-Driven Engineering places models as first-class artifacts throughout the software lifecycle requiring the availability of proper model transformation languages. Although numerous languages are available, they lack convenient facilities for debugging and supporting understanding of the transformation logic. This is not least because the underlying transformation engines operate on a low level of abstraction, hiding the operational semantics of a high-level language. Consequently, low-level debugging information is available only, e.g., variable values. To tackle these limitations, we propose a DSL on top of Colored Petri Nets (CPNs) – called Transformation Nets (TNs) – for the execution and debugging of model transformations. By integrating all artifacts of a transformation, i.e., metamodel elements, model elements, and transformation logic, a runtime model for model transformations is provided, making the afore hidden operational semantics explicit. Based on this runtime model we present various means for debugging by means of an example showing how a QVT-Relations (QVT-R) specification may be debugged using TNs.**

I. Introduction

Model-Driven Engineering (MDE) [1] proposes an active use of models to conduct the different phases of software development, whereby models are abstractions of systems and/or their environments [2]. This leads to a shift from the "everything is an object" paradigm to the "everything is a model" paradigm [3]. In the same way as programs have to follow certain syntactic constraints – commonly described by grammars – models also have to follow syntactic constraints given by so-called metamodels (MMs), which define their abstract syntax [4]. In MDE there is a recurring need to transform models between different languages and abstraction levels, e.g., to migrate between language versions, to translate models into semantic domains for analysis, to generate platform-dependent from platform-independent models, or to refine models. Model transformations always follow a certain pattern [2], as depicted in Fig. 1. In this context, the transformation definition takes place between the respective MMs using a dedicated transformation language, describing how source models should be transformed into target models. In this paper the focus is on declarative, rule-based transformation languages which only require the definition of *what* needs to be transformed by relating certain source and target MM elements. The actual specification is then executed in a transparent way by a dedicated transformation engine.

Transformation engines, however, operate on a rather low level of abstraction, appearing as black-box to the transformation designer, thereby hiding the operational semantics. As a consequence, debugging is limited to the information provided by the transformation engines. As discussed in [5], a model transformation debugger may not make use of common programming language debuggers "due to the semantic differences in abstraction between the artifacts of code and models". Consequently, a model transformation debugger must understand the model representation. Thus, the execution of a model transformation should be represented as a model, i.e., the execution should be an instance of a runtime model.

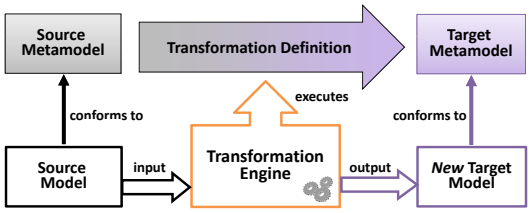

Fig. 1. Model Transformation Pattern

To tackle the limitations of current approaches and their underlying execution engines, we propose Transformation Nets (TNs), a domain-specific language (DSL) on top of Colored Petri Nets (CPNs) [6] as a runtime model for declarative model transformation languages to reveal the internals of the execution engine. First, the visual nature of TNs allows for visual debugging, e.g., to detect code-smells. Second, an explicit, model-based representation of the execution state allows to incorporate more sophisticated debugging facilities known from traditional software engineering, e.g., simulation or tracking the origin of a failure by means of reasoning backwards in time and slicing [7]. Finally, the formal properties provided by the underlying CPNs may be applied for debugging and verifying the transformation specification.

The rest of the paper is structured as follows. Section II gives an overview of the TN formalism by means of a running example. Section III to Section VI present means for debugging in TNs, ranging from visual debugging, i.e., examination of code-smells, to property-based debugging, i.e., making use of formal CPN properties for debugging. Lessons learned are presented in Section VIII. A comparison to related work is conducted in Section IX before Section X concludes.

II. Transformation Nets in a Nutshell

This section first introduces the design principles of the TN formalism. Second, we introduce a running example (cf.

This work has been funded by the FFG Bridge program under grant 832160 and by the FWF under grant P21374-N13.

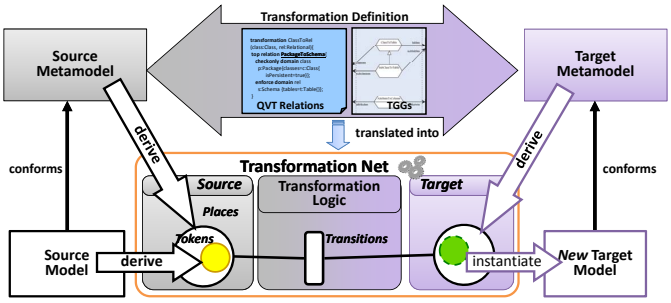

Fig. 2. Transformation Net Big Picture

Fig. 3) which is finally used to present TNs in more detail.

A. Basic Design Principles

TNs provide a runtime model for model-to-model transformations and form a DSL on top of Colored Petri Nets (CPNs) [6]. The conceptual architecture of TNs is presented in Fig. 2, showing the embedding of TNs into the model transformation pattern shown in Fig. 1. As may be seen, MM elements are represented in terms of places and model elements in terms of tokens residing in these places. Finally, for the transformation logic, which may have been specified in a dedicated transformation language, e.g., in terms of QVT-R [8] or TGGs [9], a system of transitions is derived. By the firing of transitions, tokens representing source model elements are translated into tokens representing target model elements. TNs itself are specified by means of a MM (cf. Fig. 4). The TN MM is used to specify a translation to standard CPNs in order to make use of existing execution engines and the formal properties, as discussed in detail in [10].

B. Running Example

To exemplify the TN formalism, we introduce a concrete transformation scenario that is used throughout the paper. In particular, we present a small extract of the well-known Class2Relational transformation (cf. Fig. 3) [11], which has been chosen due to its popularity. The main requirements are to translate instances of the class `Package` to equally named instances of the class `Schema` and instances of the class `Class`, which are `persistent`, to equally named instances of the class `Table`. Furthermore, it is demanded that the containment hierarchy remains.

To realize these requirements, a transformation is specified using the OMG standard transformation language QVT-R [8] (cf. Fig. 3). The specification makes use of two `relations`. The first relation `PackageToSchema` matches for `Package` instances in the source model (cf. checkonly domain in line 6 in Fig. 3) and produces equally named `Schema` instances (cf. enforce domain in line 9 in Fig. 3). The second relation `ClassToTable` follows the same design as the relation `PackageToSchema` but matches for classes, which are persistent, only (cf. condition `isPersistent=true` in line 18). Additionally, to maintain the containment hierarchy, the relation also matches for the according `Package` instance in order to contain the `Table` instance in the `Schema` instance generated for the `Package` instance. Finally, since we do not want `Schemas` containing equally named `Tables`, an according key is defined, which prohibits multiple creation of equally named `Tables`.

However, when executing the transformation specification several undesired target model elements result (cf. bold bordered elements in target model in Fig. 3), e.g., the transformation produces four `Schema` instances although the source model exhibits only two `Packages` and only one table is produced. Consequently, means for spotting the errors, i.e., debugging, are needed.

C. Transformation Nets by Example

In order to debug the above transformation, the QVT-R specification, the source and target MMs as well as the source model are transformed to an according TN, following the rules introduced above (cf. Fig. 5). For each MM element (i.e., `Classes`, `Attributes` or `References`, cf. Fig. 5) places have been derived. Second, the source model elements have been represented in terms of tokens. In particular, for every object a unicolored token is derived which is put into the corresponding place, i.e., the `Package p2` is represented

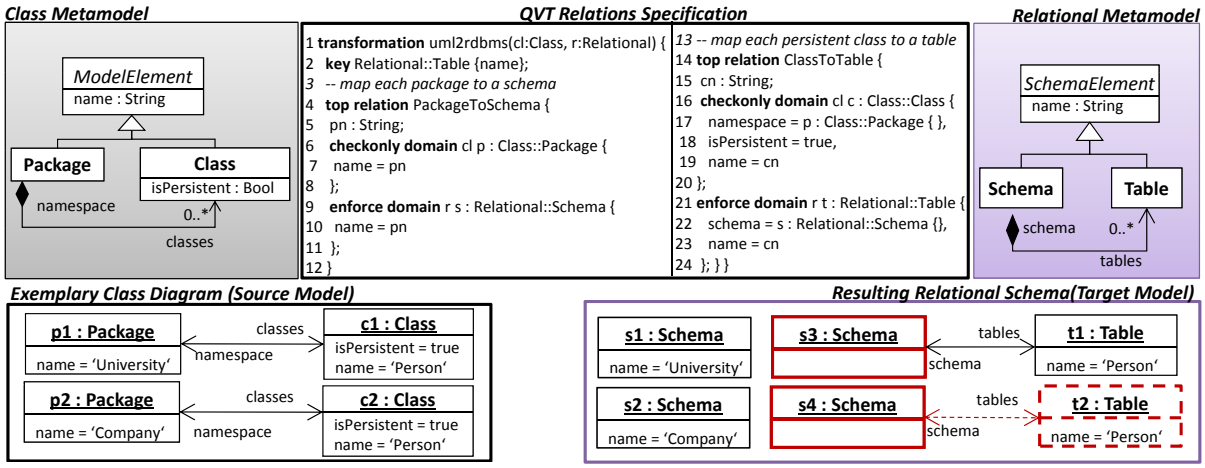

Fig. 3. Running Example

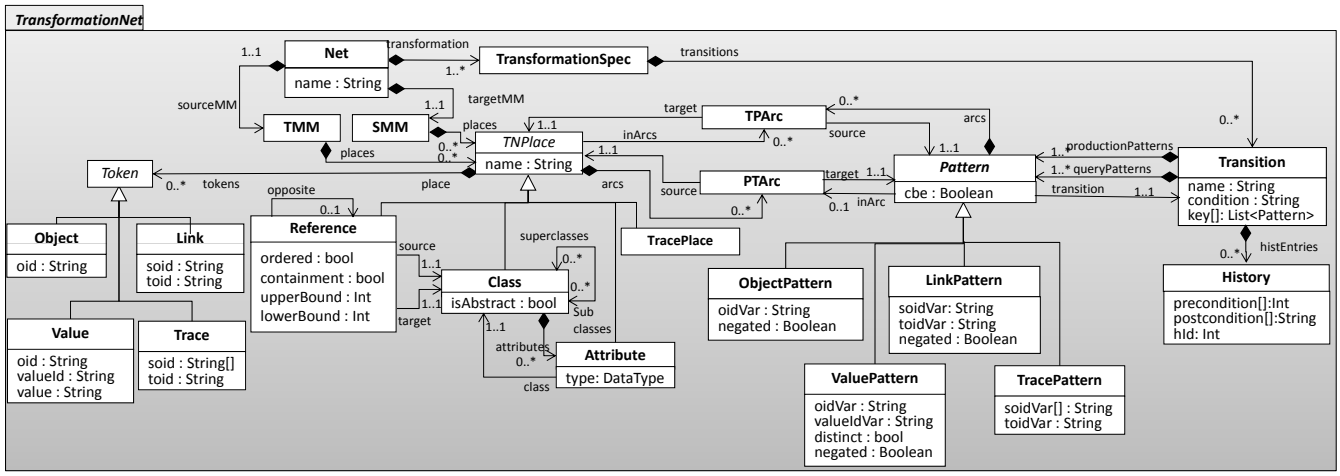

Fig. 4. Transformation Net Metamodel

by the white token in the place labeled `Package`. Values are represented by tokens consisting of two colors, whereby the upper color represents the color of the owning object, whereas the lower color encodes the actual attribute value (cf., e.g., tokens in place `ModelElement.name`). Links are again represented by two-colored tokens, whereby the outer color of the token represents the link's source object and the inner color its target object (cf,. e.g., tokens in place `classes`).

The actual transformation logic is represented in terms of transitions in TNs, which is derived from the specified transformation logic. In our example, one may see that for every QVT-R relation a transition results. Transitions consist of a left-hand side (LHS), which states the precondition that needs to be fulfilled in order to fire a transition, i.e., according tokens need to be available. The right-hand side (RHS) states the postcondition of a transition, i.e., the tokens that should be produced after firing. These conditions are expressed by means of so-called `QueryPatterns` (LHS) or `ProductionPatterns` (RHS). Thereby, again different types of patterns are provided which query/produce different types of tokens, i.e., one may distinguish `ObjectPatterns`, `ValuePatterns`, and `LinkPatterns`. Additionally, `TracePatterns` are employed to make the trace information of one transition available to dependent ones. Consequently, the elements of the `checkonly` domain of a QVT-R specification are represented in terms of `QueryPatterns` and those of the `enforce` domain are represented by according `ProductionPatterns`. In order to ensure that a target object, e.g., a `Schema`, is created only once, a QVT-R transformation engine examines the trace information and checks if the according target element has been created before, i.e., a check *before enforce semantics* is employed. QVT-R furthermore allows defining equality of objects by means of *keys*. In order to represent this behavior accordingly in TNs, every transition is marked with a check before enforce flag (cf. stereotypes on transitions in Fig. 5), i.e., target objects and their depending values or links of a transition are only created if they did not exist before. Additionally, a key may be specified for transitions by stating the according variables of the production patterns, e.g., to represent the key of our example in Fig. 3, the transition that produces `Table` instances exhibits a key containing the object itself (represented by the variable `c`) and it's attribute name (represented by the variable `cn`), shown in Fig. 5.

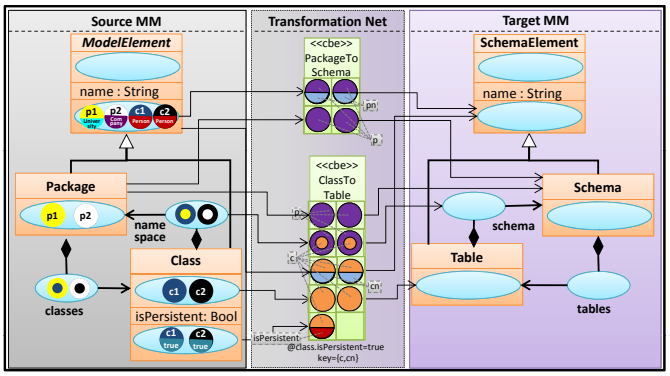

Fig. 5. TNs by Example: QVT-Relations translated to TNs

As demonstrated by the previous example a model transformation might be represented in terms of a model, i.e., the execution is an instance of a runtime model, which conforms to the TN metamodel depicted in Fig. 4.

III. Code-Smells in Model Transformations

The integrated view on model transformations of TNs allows to detect potential failures, i.e., the static structure might already indicate failures, so-called *code-smells*, which are discussed in the following.

Coherence Between Rules. Typically, rules in transformation languages interact with each other, i.e., the result produced by one rule allows other rules to transform their corresponding elements. In this respect, rules use trace information or explicit calls to synchronize each other. If several unrelated rules are specified, then they work independent of each other, potentially resulting in unconnected parts in the output model, which is normally not intended. This is especially true in the

context of Ecore-based[1] models, which demand a tree structure of the model elements. Consequently, at least a connection to the rule transforming the root container must exist.

Detection in TNs: In Transformation Nets, transitions interoperate via trace places and according `TraceQuery/ProductionPatterns` (cf. Fig. 9). Therefore, if transitions are not connected via according trace places, independent parts of the target model result. As can be seen in Fig. 5, the transitions `PackageToSchema` and `ClassToTable` are not connected to each other which is not desired since `Table` instances should be contained in their according `Schema` (cf. target model in Fig. 3).

Invalid Target Model. The generated target model of a transformation must again conform to its according MM. For example, dangling references must not occur, i.e., links have to point to a valid target object. Dangling references may occur in transformation languages if objects are deleted which are still referred by some links.

Detection in TNs: Invalid configurations of a target model may be detected by inspecting the tokens of the generated target model, i.e., the colors of links and the object color of values has to be present in the according class places. In order to ensure such well-formedness constraints, OCL invariants are added to the target places (cf. invariants `HasSource` and `HasTarget` in Listing 1). For example, a constraint is put onto reference places which checks if the according source and target objects exist. Additionally, boundedness constraints are validated, i.e., if the number of tokens that originate from a certain link token (outer color) does not exceed the specified upper-bound of the reference.

MM Coverage. If no rule matches a certain metamodel element, then this element will not participate in the transformation process and the according instances will not result in any target instances, which leads to information loss during the transformation. As on the source side, also on the target side metamodel elements may not be targeted by a single rule and therefore no according instances may be created.

Detection in TNs: If no source arc originates from a certain source place, this MM element will not be considered in the transformation. The same is true on the target side, i.e., if no arc targets a certain place representing an element of the target MM, instances of this MM element may not be created by the transformation. Both code-smells may automatically be detected by means of OCL invariants in TNs leading to according warnings (cf. invariant `SourceMMCoverage` in Listing 1 showing the invariant for the source MM).

Redundant Specification. If several rules match a certain MM element, then this may lead to redundant elements on the target side, unless according conditions match for disjoint subsets. Again, the same applies to the target MM, i.e., if elements of the target MM are targeted by several rules, it may happen that these parts will be produced several times (if no check before enforce semantics is employed).

Detection in TNs: On the source side, this code-smell may be detected if several arcs originate from one source place. On the target side, this may be detected if more than one arc targets a certain place. Again this may be automatically checked by employing OCL constraints (cf. invariant `SourceMMRedundancy` in Listing 1), leading to according warnings in TNs. Furthermore, if the TN is already executed and if a target place contains duplicates, i.e., same-colored tokens, this indicates redundantly specified parts as well.

Listing 1. OCL Invariants to detect Code-Smells

```
1  context Reference inv HasSource:
2    self.source->tokens->collect(oid)->
3    includesAll(self.tokens->collect(soid))
4  ---------------------------------------------
5  context Reference inv HasTarget:
6    self.target->tokens->collect(oid)->
7    includesAll(self.tokens->collect(toid))
8  ---------------------------------------------
9  context SMM inv SourceMMCoverage:
10   self.places->forAll(p | p.arcs->notIsEmpty())
11 ---------------------------------------------
12 context SMM inv SourceMMRedundancy:
13   self.places->forAll(p | p.arcs->size() > 1)
```

In summary, the static structure of TNs may already indicate certain failures in the transformation specification, providing a potential starting point for debugging.

IV. Simulation-Based Debugging

Although code-smells may point to failures, failures may often only be detected by means of live-debugging, i.e., simulation of the execution. The simulation of the transformation specification allows a transformation designer to get an insight into the specification, i.e., the hidden operational semantics is made explicit, in order to foster debugging. In this respect, TNs provide various means to support the transformation designer to effectively find the origin of a failure being (i) means for selecting a certain part of the transformation code, (ii) means for inspecting the current execution state and, finally, (iii) means for investigating the dynamic behavior.

A. Selection

TNs allow (i) to select an enabled transition and according bindings, i.e., debugging of the matching process, and (ii) to set breakpoints on different elements (i.e., transitions, places, tokens), to provide flexible means to select a certain starting point for debugging.

Debugging of the Matching Process. Since transformation engines may select applicable rules non-deterministically, the debugging environment needs to accordingly visualize the rules that are currently applicable. Since TNs build a DSL on top of CPNs, which support non-determinism inherently, the enabled transitions have to be made explicit, e.g., by highlighting enabled transitions (cf. transitions with a bold border in Fig. 6), indicating that (only) these transitions may be fired. In the example in Fig. 6, both transitions are enabled, since there are `Package` instances available that enable the transition `PackageToSchema`. Furthermore, the transition `ClassToTable` is enabled since also instances of persistent `Classes` are available that are contained in a `Package`.

If a transition is enabled, it may be the case that there exist several valid bindings, i.e., several combinations of

[1]Ecore is the Eclipse realization of the EMOF standard

tokens that satisfy the precondition of a transition. Thus, the transformation designer should be enabled to select a desired one. TNs support this scenario by two different mechanisms being (i) selection of calculated bindings and (ii) user-defined bindings. Concerning the first mechanism, every enabled transition may be asked for its currently possible valid bindings, which are presented to the transformation designer and from which he is allowed to select one, as can be seen in ① in Fig. 6. Concerning the second mechanism, the transformation designer may drag and drop tokens from source places to a query pattern of the according transition. The transition checks, if the specific token is part of a valid binding. If this is the case, the query pattern is bound to the according token, restricting the possible valid bindings.

Breakpoints. Whereas in programming languages breakpoints are set to the desired line of code, in TNs breakpoints may be set on (i) transitions, (ii) tokens and (iii) places. Breakpoints on transitions are closest to those known from programming languages. Per default, the execution of TNs stops at this kind of breakpoint every time the according transition is enabled. Nevertheless, the transformation designer might change this behavior and may configure the breakpoint such that it stops execution every time a certain transition is *not* enabled. If a breakpoint is attached to a certain token, execution is stopped, if this token is successfully bound to a transition, i.e., if it is part of a valid binding (cf. ② in Fig. 6). Finally, concerning places, execution is stopped either if a token is about to be read from a certain source place or if a token is going to be put into a certain target place.

To further restrict the applicability of breakpoints, conditional breakpoints are provided, i.e., OCL expressions can be attached to breakpoints. Conditional breakpoints may be specified at different levels of granularity. Thus, it may not only be defined that execution should stop, e.g., if a certain token is streamed into a certain place, i.e., a *local condition*, but also if a certain combination of tokens occurs in several different places, i.e., a *global condition*. An example for the first case is shown in ③ in Fig. 6, e.g., the conditional breakpoint attached to the place `Schema` will stop execution only if more than one `Schema` instance is produced. An example for the latter case is shown in ④ in Fig. 6 by the breakpoint attached to the place `Table`, which stops execution if the `Table` place contains more than one element and if additionally the `Schema` place contains two or more elements.

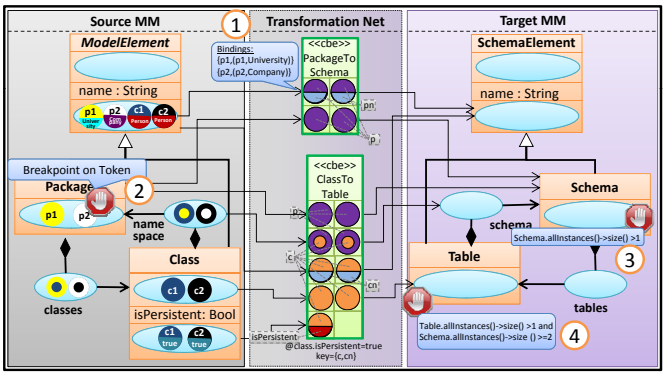

Fig. 6. Debugging Support in the Selection Phase

B. Inspection

A natural prerequisite for reasoning about the state of execution is to provide appropriate inspection mechanisms. In the following it is discussed, how the actual state of execution and the control flow is represented in TNs.

State inspection. TNs provide an integrated view on the transformation specification, i.e., not only the transformation logic itself is represented, but also the involved source and target MMs as well as their according model elements. Thus, the actual state of the transformation is explicitly presented to the transformation designer at any time during the transformation.

Visualization of control flow. On the one hand, the visualization of the control flow is achieved by highlighting the transitions ready to fire. On the other hand, the history of transitions (which is hidden per default, but may be made explicit by the transformation designer – cf. ① and ② in Fig. 7) as well as the trace tokens provide visual information on which source tokens have been transformed into which target tokens. In order to make this information even more explicit, interrelationships between tokens are highlighted on mouse-over. For example, when moving the mouse over a source object token, the relationship to according value and link tokens that are contained by this object as well as already transformed tokens that originate from the source object token are highlighted by means of dashed lines.

C. Dynamics.

A transformation designer may investigate the dynamic semantics of the transformation specification by a stepwise firing of transitions. Thus, it is possible to follow which source element is transformed into which target element, i.e., the operational semantics is made explicit. Combined with the means for selecting a certain element in the matching phase, it may easily be figured out, e.g., why a certain element is not matched by a certain transition. In the example in Fig. 6, one can see that both transitions may be fired, i.e., they might be executed in a nondeterministic order. However, since `Class` instances should only be transformed if their owning package has already been transformed, this indicates a missing interdependence between the transitions.

In summary, TNs support debugging on the model level rather than on the code level, as typically provided by current debuggers integrated in model transformation languages, i.e., they reuse the debugging features of the underlying transformation engine, only.

V. Query-Based Debugging

Debugging suffers from the well-known problem that programs execute forward in time whereas programmers must reason backwards in time to find the origin of a bug [7]. In this respect, the transformation designer needs to carefully approach the moment when the actual infection is observable

in the transformation specification. However, this is not necessarily the point, where the infection has been introduced. Thus, the transformation designer has to restart debugging and try to find the failure at some earlier point in time of execution. This is typically cumbersome and therefore means are needed that allow to reason backwards in time. Thereby, questions like "where does this target element come from" should be answered, i.e., *query-based debugging* should be enabled. The investigation of dependencies of a concrete program run is called *dynamic slicing* [7], i.e., extracting those parts of the transformation specification that deal with a certain element.

A. Dynamic Slicing by Means of OCL

Since the execution of a transformation is stored as a model, which conforms to the TN MM, OCL queries may be employed to realize dynamic slicing in order to enable backwards debugging in time for model transformations.

When inspecting the generated target model in Fig. 7, at first sight it remains unclear, why two `Schemas p1` and two `Schemas p2` get created and which transition created the according tokens, since both transitions target the place `Schema`. Therefore, a token might be asked for the transition that created the token by means of the OCL function `getCreator()` which is depicted in Listing 2. Thereby, the function first gets its according place and collects the transition (cf. line 2). Afterwards, the history of the transitions is checked whether it contains the history-id (`hId`) of the according token in its postconditions.

Listing 2. GetCreator Function

```
1 context Token: def getCreator(): Transition =
2 self.place.inArcs->collect(a:TPArc | a.source.transition)
3   ->select(histEntries)
4   ->collect(postcondition)->includes(self.hId))
```

When investigating the first token, which is labeled `p2` (cf. Step 1 in Fig. 7), one can see that it exhibits the `hId` 2. When calling the `getCreator()` function the transition `PackageToSchema` is returned, since the history-id is contained in the history of the transition. It is now possible to navigate further backwards in time by, e.g., calling the function `getInputTokens(t)`, which delivers the tokens that enabled the transition in accordance to a certain token `t` that was produced by this transition. For example it is possible to get the `Package` token that enabled the transition and that produces the target token `p2` by selecting the second token returned by the `getInput` function (cf. Step 2 in Fig. 7). Furthermore, if we execute the queries on the second target token labeled `p2` (cf. Step 3 and 4 in Fig. 7) one can see that we again obtain the same source `Package` instance which was matched and transformed twice. In order to correct the transformation it has to be ensured that the `Schema` instance produced by the transition `PackageToSchema` is reused in the transition `ClassToTable`, i.e., the transition `PackageToSchema` must produce trace information that may be queried by the transition `ClassToTable`.

Listing 3. GetInputTokens Function

```
1 context Transition: def getInputTokens(t: Token):
2   Sequence{Token} = --for object tokens only
3   self->collect(queryPatterns).inArc.source.tokens()
4     ->flatten()->select(x:Token |
5       (self->collect(histEntries)->flatten()
6       ->select(h: History | h.postcondition->
7       includes(t.oid)).precondition)->
8       includesAll(x.oid))->asSequence()
```

In summary, TNs provide means for backwards in time debugging by employing predefined OCL functions. Additionally, custom OCL functions may be used, since the execution of a model transformation is again represented as a model, i.e., the runtime model may be queried by arbitrary OCL expressions, which allows for flexible means of query-based backwards in time debugging.

VI. Property-Based Debugging

Since TNs form a DSL on top of CPNs the formal properties thereof may be applied for debugging, allowing for *property-based debugging*. For this, the state space of the underlying CPN has to be constructed to calculate diverse behavioral properties. The basic idea of state spaces is to calculate all reachable states (markings) and state changes (occurring binding elements) of the CPN. In the resulting directed graph,

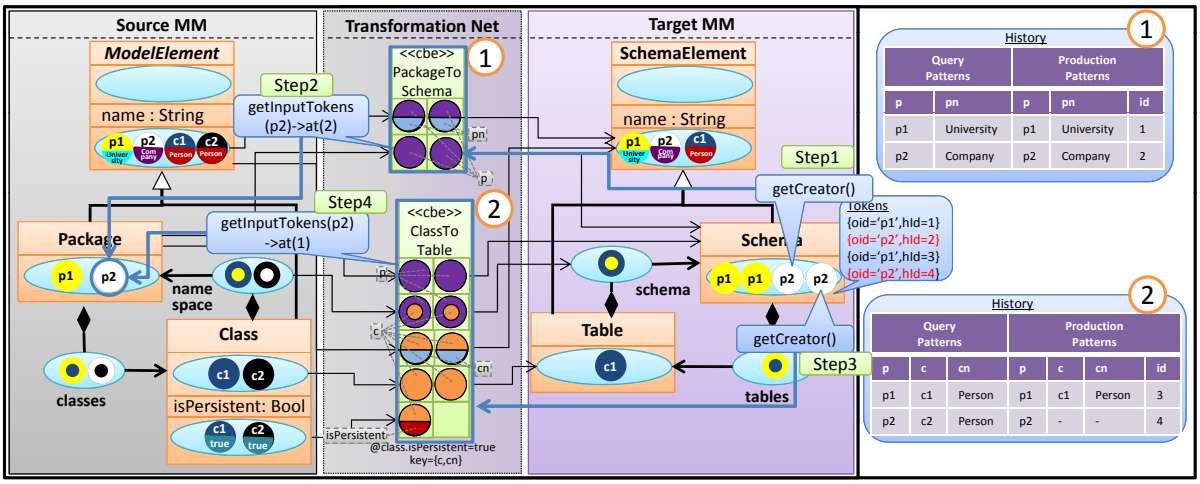

Fig. 7. Backwards in Time Reasoning in Transformation Nets

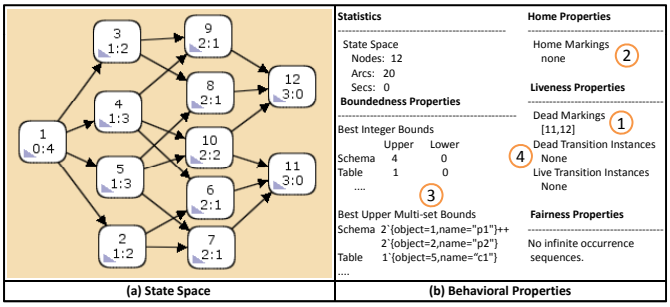

Fig. 8. State Space and Behavioral Properties of Running Example

nodes correspond to the set of reachable markings and the arcs correspond to occurring binding elements. Consequently, a state space depends on a specific initial marking, i.e., the according source model in TNs (cf. Fig. 8(a), showing the state space of the running example). State spaces provide powerful means to analyze the specified CPN and may be created fully automatically. As might be imagined, the main drawback of calculating the state space is the so-called state space explosion problem [12], i.e., the size of the state space gets too large to be stored in memory. This is mainly due to the non-deterministic selection of possible bindings of a transition and the non-deterministic selection of enabled transitions. Thus, means for state-space reduction are needed to employ these techniques, e.g., on-the-fly verification, or symmetry method [6].

The state space is used to calculate general properties on model transformations, e.g, termination or confluence. In the following, it is shown how such properties (cf. [13] for an overview) may be used to enable property-based debugging of model transformations (cf. Fig. 8(b)).

Termination and Confluence Verification using Dead and Home Markings. In a model-to-model transformation scenario (which is the focus of this paper), a model transformation is required to terminate. Thus, the state space needs to contain at least one *Dead Marking* [13], which is a state without any successors, i.e., $\exists M$ such that $M_0 \overset{\sigma}{\Rightarrow} M$ and $enabled(M) = \emptyset$, meaning that after a certain firing sequence, starting from the initial marking M_0, a marking M is reached, where no bindings are enabled any more.

Nevertheless, in order to formally verify termination, it has to be ensured that a dead marking is always reachable. For this, *home properties* are provided in CPNs, whereby a *Home Marking* M_{Home} is a marking, which may be reached from any other reachable marking, i.e., $\forall M | M \overset{\sigma}{\Rightarrow} M_{Home}$ [13]. As stated in [6], this means that "it is impossible to have an occurrence sequence starting from M_0 which may not be extended to reach M_{Home}". Consequently, if the state space contains a single *Dead Marking* which is equal to a single *Home Marking*, i.e., both states offer the same id, it is ensured that the CPN (and thus the according TN) always reaches a dead marking leading to a confluent CPN, i.e., there exists a unique terminal marking, that may always be reached. Formally this is denoted as *if* $\forall M, M' M_0 \overset{\sigma}{\Rightarrow} M \wedge enabled(M) = \emptyset \wedge M_0 \overset{\sigma}{\Rightarrow} M' \wedge enabled(M') = \emptyset$ *then* $M = M'$.

Since the calculated properties depend on the actual source model, in general the transformation would have to be tested with all possible input elements, which is not feasible. Thus, the question arises, in which situations a non-confluent behavior may occur at all, i.e., when does a transformation contain more than one home or dead state. In general, non-confluence in model transformations may occur if two rules are non-parallel independent [14]. The same is true for CPNs if the specified net is not *persistent*. A CPN is said to be persistent if "for any two enabled transitions, the firing of one transition will not disable the other one" [13], which is equal to the definition of parallel-independence. Consequently, this property has to be ensured for TNs as well.

Considering our running example, it can be seen that the state space exhibits two dead states and no home marking (cf. ① and ② in Fig. 8(b)). This is due to the fact that it can not be ensured whether `Class c1` is created (dead state 11) or `c2` (dead state 12), due to the ambiguous key defined for `Table` instances.

Model Comparison using Reachability and Boundedness Properties. To achieve a correct transformation result, an equal *Home Marking* and *Dead Marking* is a necessary, but not a sufficient condition, as it may not be ensured that this marking represents the desired target model (which has to be provided by the transformation designer in terms of an Ecore 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 may be created with the transformation logic.

If the desired target model is not reachable, a possible step to debug the transformation specification is to compare the target model generated by the transformation to an expected target model. To identify wrong or missing target elements in terms of tokens automatically, *Boundedness* properties [13] (*Integer Bounds* and *Multiset Bounds*) may be applied (cf. ③ Fig. 8(b)). *Upper integer bounds* state how many tokens reside in a certain place at most, i.e., in a first step only the number of tokens may be compared. Since no tokens are consumed from a place in TNs, the number of tokens in a place representing a target MM element has to be equal to the number of tokens derived from a desired target model. On the one hand, if there are too few tokens, the according place is highlighted to give the transformation designer a hint for debugging. On the other hand, if there are too many tokens, the according tokens may be identified by using the *Multiset Bounds*, which contain the respective marking. These tokens are then highlighted in the according TN and the transformation designer might then make use of query based debugging mechanisms to actually discover the origin of the failure. In our example, we would thus highlight the tokens `p1` and `p2` in the place `Schema` since only two tokens should be available. Additionally, the places `Table`, `schema` and `classes` would be highlighted

since they miss tokens.

Transition Error Detection using Liveness Properties. In TNs the situation might occur that a certain transition specifies a condition that is never fulfilled during the whole transformation process, i.e., the transition never fires. This situation may be detected by means of so called *Dead Transition Instances* or *L0-Liveness* [13]. Dead transition instances may be found in the state space report, whereby `none` means that no transition exists, which has never fired (cf. ④ in Fig. 8(b)). If a transition did not fire, it indicates that the source model did not enable the transition and therefore, either the specified test model did not consider a certain scenario, the specified transition is incorrect, e.g., a too restrictive condition, or even the source model is incorrect. In case of dead transition instances, the according TN transition gets highlighted in order to set the focus for debugging.

VII. Fixing Failures

The last phase in the debugging process is to actually correct the defect. In TNs, the transformation designer is allowed to (i) alter the according model, i.e., tokens may be added, edited or removed, and (ii) to change the specified transformation logic.

A. Adapting the Model

Since in TNs the model is explicitly represented by means of tokens, it is possible to add, edit or delete certain tokens in order to fix a defect during debugging. Adding tokens to places is allowed only if the according place is either a place representing a source MM element or a trace place. Adding tokens to trace places may be useful in order to continue debugging, if the transformation terminated unexpectedly. By this, questions like "would this transition fire, if there would be an according trace token" may easily be answered. Editing or deleting tokens is more complex than adding tokens, since those tokens might already have been matched by transitions. Thus, it may be the case that a transition might not have fired, if the token was not present or exhibited some different value. If a token is changed or deleted, the according history entries of the transitions have to be deleted from the transition's history in order to allow to re-execute transitions. Additionally, also the produced tokens and the history of dependent transitions have to be updated. Finally, changes or deletions might also lead to dependent changes, i.e., if an object token is deleted, all dependent value and link tokens are deleted as well, to maintain a valid model.

Finally, a remaining question is, if the changes in the model should be local to the debugging environment, i.e., the changes in the source model should not be made persistent, or if the changes during debugging should be made persistent. On the one hand, if the changes are local to the debugging environment, the transformation designer is allowed to "play around" with certain model configurations without changing the test input model. On the other hand, changes in the debugging environment probably have to be repeated in the test input model, if errors have been detected in the model. Therefore, in TNs the changes are local to the debugging environment per default, but the transformation designer may explicitly commit the changes in order to persist them.

B. Adapting the Transformation Logic

The transformation logic represented in TNs may be changed as well, i.e., it is allowed to add or edit existing transitions or trace places during debugging. Furthermore, it is allowed to edit existing transitions, e.g., by adding further query tokens or deleting a production token. In any of these cases, the history of the according transition as well as those of dependent transitions have to be updated, in order to allow to re-evaluate the according parts of the transformation specification. Nevertheless, since TNs provide a runtime model for declarative model-to-model transformation languages, i.e., it is possible to represent the actual transformation logic in terms of TN concepts, the back propagation of changes in the transformation logic to the actual transformation languages represents a major challenge. Currently, the back propagation requires the specification of an explicit transformation, i.e., not only a transformation from the transformation language to TNs is required, but also a transformation from TN concepts to the concepts of a specific transformation language, e.g., a transformation from TNs to QVT-R.

As discussed before, there are two failures in the running example. First, the transitions are independent from each other resulting in too many `Schema` instances and second the specified key for `Table` instances is ambiguous. To fix the first failure, a trace place is introduced (cf. ① in Fig. 9). Additionally, the transition `PackageToSchema` adds a `TracePattern`, stating which `Package` instances have been translated to which `Schema` instances. This trace information is then queried by a `TracePattern` in the transition `ClassToTable`. In this respect, the already produced `Schema` instance is reused to set the references. Consequently, the resulting target model consists of two `Schema` instances only, which contain the `Table` instances. The changes in TNs are represented in QVT-R by adding an according when-clause in the relation `ClassToTable`. Thus, this relation is executed only if it is also possible to execute the relation `PackageToSchema`. Concerning the second failure, the reference to the `Schema` needs to be added to the key to ensure uniqueness (cf. ② in Fig. 9).

VIII. Lessons Learned

This section presents lessons learned and thereby discusses key features as well as current limitations of the TN approach.

Visual Syntax and Live Programming Fosters Debugging. TNs provide a visual formalism for defining model transformations, which is especially favorable for debugging purposes. This is not least since the flow of model elements undergoing certain transformations can be directly followed by observing the flow of tokens, whereby undesired results can be detected easily. Another characteristic of TNs that fosters debuggability, is live programming, i.e., some piece of transformation logic can be executed and tested immediately after definition without any further compilation step.

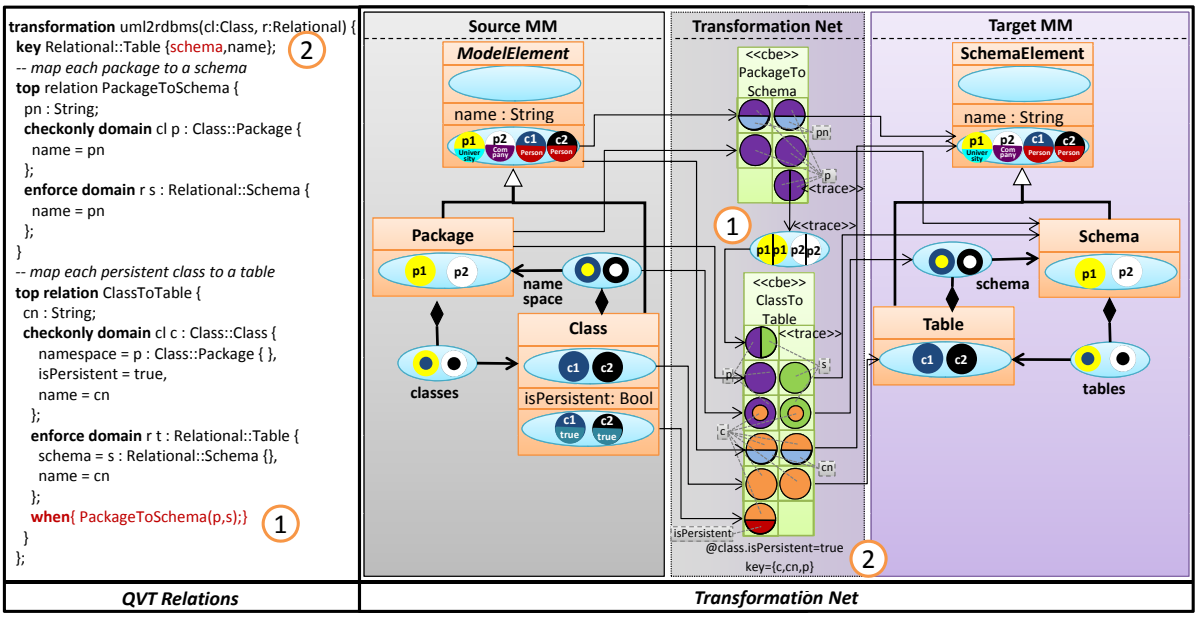

Fig. 9. Bug Fixing in Transformation Nets

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 state space is constructed in our prototype to calculate properties leading to memory and performance problems for large source models and transformation specifications. A detailed comparison of different reduction mechanisms and their usefulness in TNs is required in order to identify those algorithms that work best or how existing approaches might be adapted in order to fit to the domain of model transformations.

Concurrency in Petri Nets Allows Parallel Execution. 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 this respect, independent parts of a transformation could be executed in parallel, making TNs suitable as efficient execution engine. Thereby, the properties Home State and Dead Markings can ensure confluence, even in case of parallel execution.

IX. Related Work

Related work regarding debugging support of transformation languages and approaches for verifying properties of model transformations are presented in the following.

Debugging Support of Transformation Languages. In general, there is little debugging support for transformation languages. Most often only low-level information 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 [15] compiles TGG rules [16] into Fujaba story diagrams that are implemented in Java, which obstructs a direct debugging on the level of TGG rules. In the approach proposed by Geiger [17], 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 [18] produce debug reports that trace an execution, only, but do not provide interactive debugging facilities. Although the ATL debugger [19] 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 transformation, e.g., the coherence between rules. SmartQVT and TefKat [20] allow for similar debugging functionality.

Hibberd et al. [21] 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. 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.

Properties of Model Transformations. Current transformation languages provide only limited support to debug transformation specifications by means of properties. However, several approaches exist that translate the specification to an external formalism in order to verify certain properties. 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 [22] translates individual graph rules into a Place/Transition Net to check for its termination. Another approach is described in [23], 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. De Lara and Guerra [24] proposes to translate QVT-R into CPNs – on the one hand to provide a formal semantics for QVT-R and on the other hand to analyze QVT-R specifications – pursuing similar ideas as followed in our previous work [25], [26]. Nevertheless, these approaches are using Petri Nets as a back-end for analyzing properties of transformations only, whereas we are using a DSL on top of CPNs as a front-end for model transformations, thereby fostering debuggability. Besides using Petri Nets for analysis, researchers apply OCL (e.g., [27]) or formal languages like Maude [28] for analysis.

X. Conclusion and Future Work

In this paper we presented several means for debugging model transformation based on the TN formalism, being a DSL on top of CPNs. In order to find the origin of a failure, means for live-debugging (support in the matching phase, simulation, breakpoints) as well as backwards-in-time debugging by means of OCL queries on the TN runtime model have been presented. Finally, it was discussed how CPN properties may be applied for debugging.

The TN formalism provides a runtime model for declarative model-to-model transformation languages. However, since many other scenarios are possible as well, we are investigating how endogenous transformations may be incorporated into the TN formalism and to which respect the TN formalism is suitable to debug imperative transformation languages. Thereby, we also plan to make the TN runtime model transparent to the user, i.e., the debugging features should be integrated into the according transformation languages. Additionally, we plan to focus on state space reduction mechanisms, e.g. by using the temporal logic LTL.

References

[1] D. C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer*, vol. 39, February 2006.

[2] K. Czarnecki and S. Helsen, "Feature-based Survey of Model Transformation Approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[3] J. Bézivin, "On the Unification Power of Models," *Software and System Modeling*, vol. 4, no. 2, p. 31, 2005.

[4] T. Kühne, "Matters of (meta-)modeling," *Software and System Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[5] Y. Lin, J. Zhang, and J. Gray, "A Testing Framework for Model Transformations," in *Proc. of Model-Driven Software Development - Research and Practice in Software Engineering*, 2005, pp. 219–236.

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

[7] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging – Second Edition*. Morgan Kaufmann, 2009.

[8] OMG, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," http://www.omg.org/spec/QVT/1.1/Beta2/PDF/, 2009.

[9] A. Schürr, "Specification of Graph Translators with Triple Graph Grammars," in *Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, 1994, pp. 151–163.

[10] J. Schönböck, "Testing and Debugging of Model Transformations," Ph.D. dissertation, Vienna University of Technology, Business Informatics Group, 2011.

[11] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt, Eds., *Proc. of Model Transformations in Practice Workshop @ Int. Conf. on Model Driven Engineering Languages & Systems, Montego Bay, Jamaica*, 2005.

[12] A. Valmari, "The State Explosion Problem," in *Proc. of Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, based on the Advanced Course on Petri Nets, September, Dagstuhl, Germany*, 1998, pp. 429–528.

[13] T. Murata, "Petri nets: Properties, Analysis and Applications," *Proc. of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[14] R. Heckel, J. M. Küster, and G. Taentzer, "Confluence of Typed Attributed Graph Transformation Systems," in *Proc. of the 1st Int. Conf. on Graph Transformation,Barcelona, Spain*, 2002, pp. 161–176.

[15] R. Wagner, "Developing Model Transformations with Fujaba," in *Proc. of the 4th Int. Fujaba Days, Bayreuth, Germany*, 2006, pp. 79–82.

[16] A. Königs, "Model Transformation with TGGs," in *Proc. of Model Transformations in Practice Workshop @ Int. Conf. on Model Driven Engineering Languages & Systems, Montego Bay, Jamaica*, 2005.

[17] L. Geiger, "Model Level Debugging with Fujaba," in *Proc. of 6th Int. Fujaba Days, Dresden, Germany*, 2008, pp. 23–28.

[18] A. Balogh and D. Varró, "Advanced model transformation language constructs in the VIATRA2 framework," in *Proc. of ACM Symposium On Applied Computing, Dijon, France*, 2006, pp. 1280–1287.

[19] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A Model Transformation Tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, June 2008.

[20] M. Lawley and J. Steel, "Practical Declarative Model Transformation with Tefkat," in *Proc. of Model Driven Engineering Languages & Systems Satellite Events, Montego Bay, Jamaica*, 2006, pp. 139–150.

[21] M. T. Hibberd, M. J. Lawley, and K. Raymond, "Forensic Debugging of Model Transformations," in *Proc of Int. Conf. on Model Driven Engineering Languages & Systems, Nashville, USA*, 2007, pp. 589–604.

[22] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer, "Termination Analysis of Model Transformation by Petri Nets," in *Proc. 3rd Int. Conf. on Graph Transformations, Natal, Rio Grande do Norte, Brazil*, 2006, pp. 260–274.

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

[24] J. Lara and E. Guerra, "Formal Support for QVT-Relations with Coloured Petri Nets," in *Proc. of Int. Conf. on Model Driven Engineering Languages & Systems, Denver, CO, USA*, 2009, pp. 256–270.

[25] M. Wimmer, A. Kusel, J. Schönböck, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets," in *Proc. of Int. Conf. on Model Driven Engineering Languages & Systems, Denver, CO, USA*, 2009, pp. 727–732.

[26] M. Wimmer, G. Kappel, J. Schönböck, A. Kusel, W. Retschitzegger, and W. Schwinger, "A Petri Net based Debugging Environment for QVT Relations," in *Proceedings of Int. Conf on Automated Software Engineering, Auckland, New Zealand*, 2009, pp. 1–12.

[27] J. Cabot, R. Clarisó, E. Guerra, and J. Lara, "Analysing Graph Transformation Rules through OCL," in *Proc. of the Int. Conf. on Model Transformations, Zürich, Switzerland*, 2008.

[28] J. Troya and A. Vallecillo, "Towards a rewriting logic semantics for ATL," in *Proc. of Int. Conf. on Model Transformations, Malaga, Spain*, 2010, pp. 230–244.