Transformation Nets - A Runtime Model for Transformation Languages*

Johannes Schoenboeck

Institute of Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
`schoenboeck@big.tuwien.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 not least because transformation engines operate on a low level of abstraction, hide the operational semantics of a transformation and scatter metamodels, models, transformation logic, and trace information across different artifacts. To tackle these problems, we propose a DSL on top of Colored Petri Nets (CPNs)—called Transformation Nets—for the development, 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, and ensures a homogenous view on transformations by representing them on the basis of the runtime model.

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

1 Introduction and Problem Description

The availability of proper model transformation languages is *the* crucial factor in MDE, since transformation languages are as important for MDE as compilers are for high-level programming languages. Several kinds of dedicated transformation languages have been proposed (see [1] for an overview), comprising imperative, declarative and hybrid ones. Imperative approaches allow to specify complex transformations more easily but induce more overhead code as many issues have to be accomplished in an explicit way, e.g., specification of the execution order. 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, transformation engines used for executing model transformations operate on a considerably lower level of abstraction than the transformation logic itself. This leads to an impedance mismatch between specification and execution, thus hampering understandability and debuggabilty. Second, declarative and hybrid approaches use black-box transformation engines hiding the actual operational semantics, e.g., the Atlas Transformation Language (ATL) uses a stack

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

machine [2]. As a consequence, debugging of model transformations is limited to the information provided by the transformation engine, 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 or not. Finally, comprehensibility of transformation logic is hampered as current transformation languages provide a limited view on the execution of model transformations, since metamodels, models, transformation specification, and trace information are scattered across different artifacts.

What is needed is a declarative approach that integrates all artifacts in a common view thereby providing a runtime model that makes the operational semantics of a transformation specification explicit. Based on this runtime model, debugging on the level of transformation specifications should be enabled rather than just forcing transformation designers to interpret low-level error messages.

2 Proposed Solution

The conceptual architecture of our approach tackling the aforementioned limitations is shown in Fig. 1. The Transformation Net formalism [3], a DSL on top of CPNs [4], follows a process-oriented view towards model transformations making the operational semantics of the transformation logic explicit. Transformation Nets form a runtime model that provides 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.

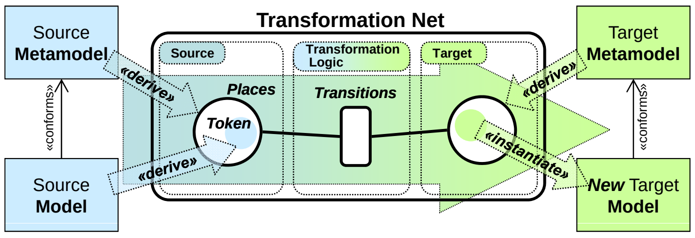


Fig. 1. Conceptual Architecture of Transformation Nets

Furthermore, Transformation Nets provide a uniform formalism not only for representing the transformation logic together with the metamodels and the models themselves, but also for executing the transformations. In particular, places in Transformation Nets are derived from elements of metamodels, whereby a place is created for every class, attribute and reference in a metamodel. 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 finally representing instances of the target metamodel to be created and thereby establishing trace information in terms of tokens within trace places. The abstract syntax of the Transformation Net language is formalized by means of a metamodel (see [3]) conforming to the Ecore meta-metamodel, the Eclipse realization of OMG's MOF standard.

3 Expected Contributions

By the proposed solution we expect three main contributions: (1) a runtime model based on CPNs being the prerequisite for both, (2) debugging of transformation languages and (3) an environment to specify and to debug Transformation Nets.

Runtime Model for Model Transformation Languages. The runtime model based on CPNs allows transformation designers to gain an explicit, integrated representation of the semantics of model transformations which particularly favors debugging and understanding. The runtime might act as a transformation engine for various declarative transformation languages, e.g., QVT Relations, to benefit from our debugging features. As Petri Nets provide formal definitions of concurrent operations, parallel execution of transformation logic is possible to increase efficiency of the execution phase. To ensure valid target models it should be possible to specify different levels of integrity constraints, i.e., an optimistic approach, where conformance will be checked after transformation or a pessimistic approach, where conformance is ensured during transformation.

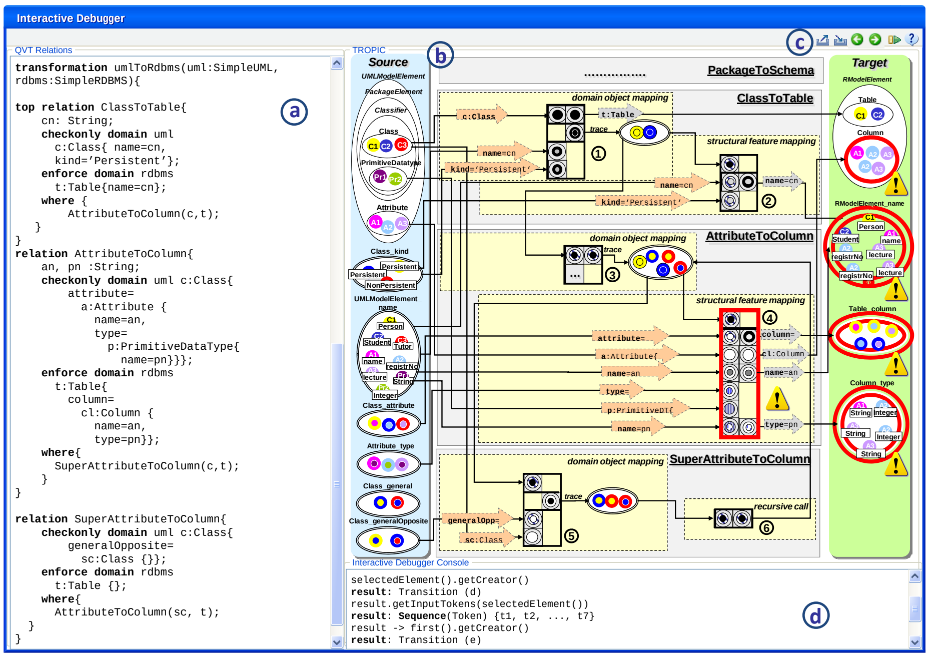
Debugging of Model Transformations. The runtime model supports transformation designers in debugging transformation logic along the three main phases of debugging: (1) observing facts, (2) tracking origins and (3) fixing bugs. Observing facts and tracking origins can be achieved using appropriate mechanisms before (i.e., static debugging), during (i.e., life debugging) or transformation execution (i.e., forensic debugging).

Observing facts. Formal properties of CPNs [5] such as *Reachability*, *Liveness* or *Persistence* can be exploited for *static debugging. Reachability* allows to check if the desired final state (i.e., the expected output model) is reachable from the initial state (i.e., the given input model) with the defined transformation logic. *Liveness* properties can be applied to detect "dead" transformation logic (*L0-liveness*) or for defining test cases, e.g., to check if a transition fires as many times as expected (*L2-liveness*). Finally, the *persistence* property can be used to detect non-determinism or erroneous race conditions. Besides *live debugging* (simulation) also *forensic debugging* is supported in that the resulting target model can be compared to an expected target model to identify wrong target tokens similar to unit-based testing of software.

Tracking origins. The transformation process can be executed stepwise revealing which tokens enable a certain transition and which tokens get produced by firing this transition, enabling *live debugging.* This is possible because Transformation Nets provide a white-box view on model transformation execution, i.e., the specification does not need to be translated into some low-level executable artifact but can be executed right away. Additionally, the runtime metamodel also allows to employ MDE standards for debugging such as OCL to define conditional breakpoints or to explore the execution state by using queries on the runtime model to reason backwards in time. Additionally, *forensic debugging* is enabled by tokens in the corresponding trace places indicating which source elements were used to create specific target elements.

Fixing Bugs. Since model transformations can be "simulated" on the basis of CPNs, a bug can be fixed right away without interruption of the simulation.

Development Environment for Transformation Nets. The runtime model as well as the debugging techniques will be integrated in a development environment supporting the creation, execution and debugging of Transformation Nets. We will additionally provide mappings from declarative transformation languages to Transformation Nets for debugging purposes, e.g., for QVT Relations [6] as shown in Fig. 2. The editor toolbar provides common debugging functionalities such as enabling stepwise debugging to figure out the operational semantics by firing transitions including an undo/redo mechanism. Besides these standard debugging functionalities, there are additional debugging features which result as a benefit of using a dedicated runtime model, e.g., an *Interactive OCL Console* to explore and to understand the history of a transformation by determining and tracking paths of produced tokens [7].



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

Fig. 2. Development Environment for Transformation Nets.

4 Related Work

Related work regarding the use of Petri Nets for model transformations and debugging support of transformation languages is presented in the following.

Petri Nets and Model Transformations. 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 [8] translates individual graph rules into a place/transition net and checks for its termination. Another approach is described in [9], which applies a transition system for modeling the dynamic behavior of a metamodel.

Compared to these two approaches, our intention to use Petri nets is entirely different, i.e., not just using them as a back-end for automatically analyzing properties of transformations, but additionally use them as a front-end for fostering debuggability and understandability.

Debugging Support for Model Transformations. In the Fujaba environment, a plugin called MoTE [10] compiles TGG rules [11] into Fujaba story diagrams that are implemented in Java, which obstructs a direct debugging on the level of TGG rules. Furthermore, approaches like VIATRA [12] produce debug reports that trace an execution, only, but do not allow to debug certain transformation rules. Debugging of ATL [2] is based on the step-wise execution of a stack-machine that interprets ATL byte-code. In contrast to the above language-specific debugging facilities, Hibberd et al. [13] present forensic debugging techniques by utilizing trace information of model transformations for localizing bugs. In addition, they present a technique based on program slicing for further narrowing the area where a bug might be located.

While Hibberd focuses only on forensic debugging, Transformation Nets additionally enable live debugging. What sets our approach apart from these approaches is that all debugging activities are carried out on a higher level of abstraction and on a single formalism. Current approaches do not 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 only provide snapshots of the current transformation state. Furthermore, our approach is unique in allowing interactive execution.i.e., fixing bugs during execution.

5 Plan for Evaluation

The plan for evaluating our approach builds on empirical studies and on applying case studies.

Empirical studies. To evaluate usability and applicability of Transformation Nets we intend to conduct empirical studies with students from our model engineering courses (around 200 master students every year) based on questionnaires. Additionally, we will use the debugging questions of Hibberd et al. [13] and let students answer those questions with our approach to verify the debugging support.

Case Studies. Case studies for transforming models will be set up and implemented with distinct existing model transformation languages, including Transformation Nets. The results will be evaluated on the basis of a suitable subset of the ISO 9126 software quality model [14]. We intend to use a representative selection of metamodels defining structural and behavioral languages. For this, we aim to use the well-known Class2Relational example [15] and the CSP2ActivityDiagrams example [16]. These case studies will also be used to evaluate to what extent concurrent execution of transformation logic improves the performance of model transformations.

6 Current Status

Currently a first prototype to specify and execute Transformation Nets is available which is applied in several case studies to verify the basic approach, as the research is in it's initial state. Additionally, a complementing research focuses on how Transformation Nets can be employed in reusable mapping operators [17].

References

1. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal **45**(3) (2006)
2. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming **72**(1-2) (June 2008) 31–39
3. Wimmer, M., Kusel, A., Reiter, T., Retschitzegger, W., Schwinger, W., Kappel, G.: Lost in Translation? Transformation Nets to the Rescue! In: Proc. of 3rd Int. United Information Systems Conf. (UNISCON'09), Sydney, Australia, Springer (April 2009) 315–327
4. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modeling and Validation of Concurrent Systems. Springer (2009)
5. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
6. Kusel, A., Schwinger, W., Wimmer, M., Retschitzegger, W.: Common pitfalls of using qvt relations - graphical debugging as remedy. In: 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009), Potsdam, Germany, IEEE Computer Society (June 2009) 329–334
7. Wimmer, M., Kusel, A., Schoenboeck, J., Kappel, G., Retschitzegger, W., Schwinger, W.: Reviving QVT Relations: Model-based Debugging using Colored Petri Nets. In: Proc. of MoDELS '09, Denver (2009)
8. Varró, D., Varró-Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination Analysis of Model Transformation by Petri Nets. In: Proc. of Int. Conf. on Graph Transformation. Volume 4178., Natal, Brazil, LNCS (September 2006) 260–274
9. de Lara, J., Vangheluwe, H.: Translating Model Simulators to Analysis Models. In: Proc. of 11th Int. Conf. on Fundamental Approaches to Software Engineering, Budapest, Hungary (April 2008) 77–92
10. Wagner, R.: Developing Model Transformations with Fujaba. Technical report, University of Paderborn (September 2006)
11. Koenigs, A.: Model Transformation with Triple Graph Grammars. Model Transformations in Practice Workshop of MODELS'05, Montego Bay, Jamaica (2005)
12. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: Proc. of SAC '06, NY, USA, ACM (2006) 1280–1287
13. Hibberd, M.T., Lawley, M.J., Raymond, K.: Forensic Debugging of Model Transformations. In: Proc. of MoDELS'07, Nashville, USA (October 2007)
14. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland: ISO/IEC Standard No. 9126: Software engineering Product quality; Parts 14 (2004)
15. Bézivin, J., Rumpe, B., Schürr, A., Tratt, L.: Model Transformations in Practice Workshop of MODELS'05, Montego Bay, Jamaica. http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf (2005)
16. Taentzer, Gabriele and Rensink, Arend: AGTIVE 2007 Tool Contest. http://www.informatik.uni-marburg.de/~swt/agtive-contest/ (2007)
17. Kusel, A.: TROPIC - A Framework for Building Reusable Transformation Components. Doctoral Symposium, Models 2009, Denver, USA (2009)