Reviving QVT Relations: Model-based Debugging using Colored Petri Nets*

M. Wimmer[1], A. Kusel[2], J. Schoenboeck[1], G. Kappel[1],
W. Retschitzegger[3], and W. Schwinger[2]

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

Abstract. The standardized QVT Relations language, one cornerstone of Model-Driven Architecture (MDA), has not yet gained widespread use in practice, not least due to missing tool support in general and inadequate debugging support in particular. Transformation engines interpreting QVT Relations operate on a low level of abstraction, hide the operational semantics of a transformation and scatter metamodels, models, QVT code, and traces across different artifacts. We propose a model-based debugger representing QVT Relations on bases of TROPIC, a model transformation framework which utilizes a variant of Colored Petri Nets (CPNs) providing an explicit runtime model and a homogenous view on all artifacts of a transformation.

Key words: QVT Relations, Debugging, Model Transformations, CPN

1 Introduction

In the MDA paradigm, model transformation languages play a vital role, leading already to the standardization of the Query/View/Transformation (QVT) language [1]. Especially for declarative transformation languages, such as QVT Relations, appropriate debugging facilities are of outermost importance, as is also the case for declarative languages in general, since the missing operational semantics hampers observation, tracking and fixing of bugs [2]. Existing approaches for executing and debugging QVT Relations (e.g., mediniQVT[4]) are still in its infancy [3] and often provide only low-level debugging information such as logging messages or variable values, hide the execution order of transformation rules and scatter metamodels, models, rules and traces across different artifacts.

We propose a model-based debugger [4] representing QVT Relations on bases of TROPIC (Transformations on Petri Nets in Color) [5, 6], a model transformation framework based on Colored Petri Nets (CPNs) [7], adapted to the needs of

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

[4] http://projects.ikv.de/qvt

transformation designers [8]. With this, firstly, an explicit runtime model is provided, which can be easily exploited for debugging purposes, e.g., by using OCL queries, thus representing a white-box view on the transformation. Secondly, a homogenous view on all transformation artifacts is ensured by representing them in terms of the basic CPN concepts places, tokens and transitions.

The remainder of this paper is structured as follows. Section 2 introduces the basics of QVT Relations and TROPIC as well as of the translation in between. Section 3 introduces an interactive debugging environment offering several features for model-based debugging of transformations and finally, Section 4 provides an outlook on future work.

2 QVT Relations and TROPIC at a Glance

This section briefly illustrates the main language concepts of QVT Relations and TROPIC for describing transformation logic, details their main differences on the execution level and discusses the design rationale of the translation between the language concepts.

QVT Relations. Using QVT Relations, transformation logic between two different metamodels is specified as a set of relations that must hold for the transformation to be successful. Relations contain a set of so-called *DomainPatterns* used to match for existing source model elements in order to instantiate new target model elements or to modify existing ones. During execution of a transformation by an engine (cf. left part of Fig. 1) trace information is available in order to verify the transformation result, only, leaving the full operational semantics within in a black box.

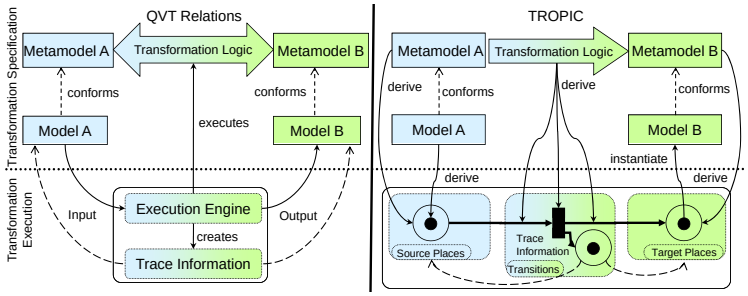

Fig. 1. Model Transformations in QVT Relations and TROPIC

TROPIC. TROPIC uses Colored Petri Net concepts [7], being mainly *places*, *tokens* and *transitions*, for the specification and execution of model transformations. In particular, places are derived from elements of metamodels, tokens from elements of models and transitions from the actual transformation logic (shown in the right part of Fig. 1). The existence of certain model elements allows transitions to fire and thus stream tokens to the target places representing instances of the target metamodel to be created and thereby establishing trace information in terms of tokens in additional places. TROPIC, thus, provides 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. Therefore, no impedance mismatch between specification and execution occurs, allowing for enhanced debuggability of model transformations.

Translation between QVT Relations and TROPIC. The translation between the concepts of QVT Relations and TROPIC has been performed on basis of their metamodels. We assume a syntactically correct QVT Relations specification since only in this case we can guarantee a correct translation to TROPIC and the propagation of changes in the transformation logic represented by TROPIC back to QVT Relations. Whereas QVT Relations only references the metamodel files, TROPIC explicitly represents each element of the metamodels as first class concept in terms of places. Regarding models, QVT Relations provides no explicit representation mechanism, which is again in contrast to TROPIC, where each model element is explicitly represented by tokens residing in corresponding places. Finally, in the textual syntax of QVT Relations the correspondences between source elements and target elements as well as the interplay among different relations are hard to grasp. TROPIC on the other hand visualizes these correspondences as well as the interplay among the relations utilizing transitions consisting of a LHS representing the pre-conditions of a certain transformation, and a RHS depicting its post-condition by means of color patterns. For further details on this translation it is referred to [9].

3 Debugging Environment for QVT Relations

Our debugging environment is based on Eclipse and includes two editors, one that presents the QVT Relations in textual syntax (cf. Fig. 2a) and another one that shows the graphical representation thereof in TROPIC (cf. Fig. 2b). The TROPIC editor toolbar (cf. Fig. 2c) provides common debugging functionalities to figure out the operational semantics such as stepwise debugging by firing transitions including an undo/redo mechanism. Furthermore, functionalities are provided to save the generated target model, i.e., to switch from the token representation to a model representation, or to load a new source model into the debugging environment.

OCL for Debugging. The utilization of a dedicated runtime model allows to employ OCL for two different debugging purposes. Firstly, OCL can be used to define conditional breakpoints at different levels of granularity, e.g., if a certain token is streamed into a certain place, or if tokens occur in several different places. Secondly, OCL can be used to tackle the well-known debugging problem that programs execute forward in time whereas programmers must reason backwards in time to find the origin of a bug. For this, a dedicated debugging console based on the *Interactive OCL Console* of Eclipse (cf. Fig. 2d) is supported, providing several pre-defined debugging functions to explore and to understand the history of a transformation by determining and tracking paths of produced tokens (exemplarily shown in Table 1).

Table 1. OCL operations for debugging

Context	QCL Debugging Operation	Description
Place	`getMatchingTokens:Set(Token)`	tokens that match a transition
	`getMismatchedTokens:Set(Token)`	tokens not matching a transition
Token	`getCreator:Transition`	transition that created a token
Transition	`getInputTokens(Token):Set(Token)`	source tokens of a transition

Debugging Phases. In the following a possible usage scenario of our debugging environment is described according to the three debugging phases, observing facts, tracking origins and fixing bugs (cf. Fig. 2).

Observing Facts. Observing facts during a certain transformation execution can be done either by simulating the transformation and watch for unexpected behavior or by debugging the transformation step-by-step. In order to detect unexpected behavior automatically, the resulting target model can be compared to an expected target model to identify wrong or missing target tokens. If such faulty parts of the target model are detected, the owning target places as well as the transitions that produce tokens in these places are highlighted to ease finding the reasons for the errors (cf. indicated by exclamation marks in Fig. 2).

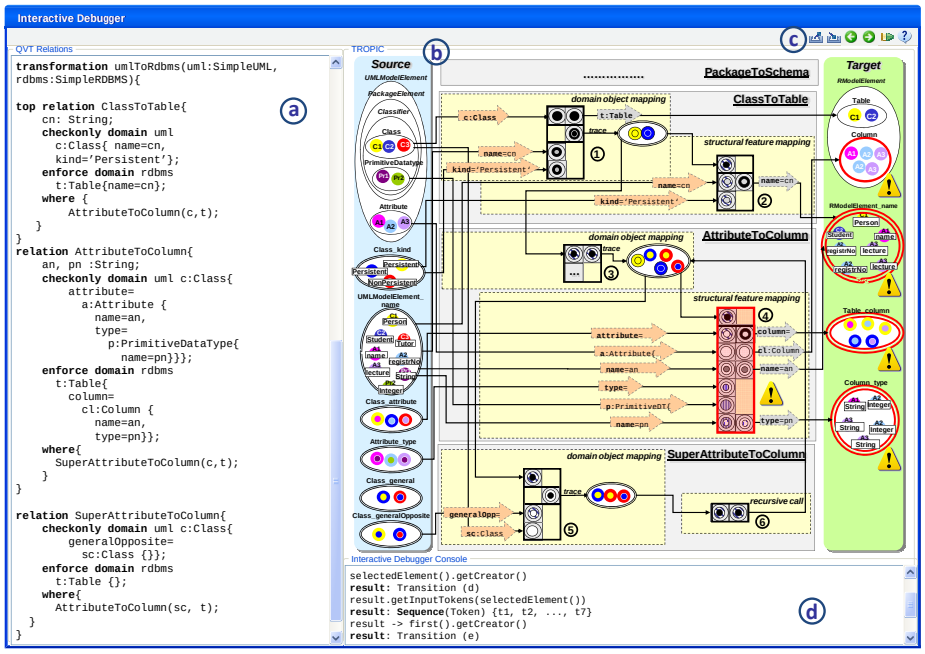

Fig. 2. Debugging Environment showing parts of the UML2Relational Example [1]

Tracking Origins. The origin of an error has to be discovered by reasoning backwards in time, questioning, e.g., why certain tokens have been created. The graphical representation in Fig. 2 shows that the tokens in question have

been created by transition 4, the source tokens responsible for creating exactly these tokens, however, are unknown. The paths of these produced tokens can be tracked back by means of our predefined OCL functions.

Fixing Bugs. After finding the origins of a bug, it is possible to adapt the transformation logic during debugging directly in TROPIC and propagate the changes back to QVT Relations.

4 Further Work

Several issues for future work remain open. As stated in [10], the QVT standard defines the operational semantics of QVT Relations twofold and only informally, firstly in natural language and secondly by a translation to QVT Core, being incompatible to each other. This situation led to different implementations of the operational semantics in different tools. Currently, our translation is based on the implementation of mediniQVT, but we are planning to investigate the implementations of different tools. Additionally, as TROPIC is based on a variant of CPNs we will explore if Petri Net properties such as persistence or liveness can be used to check for potential shortcomings in QVT Relations specifications.

References

1. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. www.omg.org/docs/ptc/07-07-07.pdf (2007)
2. Wadler, P.: Why no one uses functional languages. SIGPLAN Not. **33**(8) (1998) 23–27
3. Kurtev, I.: State of the Art of QVT: A Model Transformation Language Standard. Int. Workshop on Applications of Graph Transformation with Industrial Relevance (2007)
4. Kusel, A., Schwinger, W., Wimmer, M., Retschitzegger, W.: Common Pitfalls of Using QVT Relations - Graphical Debugging as Remedy. Int. Workshop on UML and AADL @ ICECCS'09 (2009)
5. Reiter, T., Wimmer, M., Kargl, H.: Towards a runtime model based on colored Petri nets for the execution of model transformations. In: 3rd Workshop on Models and Aspects @ ECOOP'07, Berlin (2007)
6. Wimmer, M., Kusel, A., Reiter, T., Retschitzegger, W., Schwinger, W., Kappel, G.: Lost in Translation? Transformation Nets to the Rescue! In: 8th Int. Conf. on Information Systems Technology and its Applications (UNISCON'09), Sydney (2009)
7. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modeling and Validation of Concurrent Systems. Springer (2009)
8. Wimmer, M., Kusel, A., Schoenboeck, J., Reiter, T., Retschitzegger, W., Schwinger, W.: 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)
9. Wimmer, M., Kusel, A., Schoenboeck, J., Kappel, G., Retschitzegger, W., Schwinger, W.: A Petri Net based Debugging Environment for QVT Relations. Technical report, Vienna University of Technology (2009)
10. Stevens, P.: A simple game-theoretic approach to checkonly QVT Relations. In: Proc. of Int. Conf. on Model Transformations, ICMT'09. (June 2009)