



FAKULTÄT FÜR **INFORMATIK**

Graphical Debugging of QVT Relations using Transformation Nets

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Patrick Zwickl

Matrikelnummer 0525849

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:

Betreuer: Univ.-Prof. Mag. DI Dr. Gerti Kappel

Mitwirkung: DI(FH) Johannes Schönböck

Wien, 07.12.2009

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Technische Universität Wien

A-1040 Wien Karlsplatz 13 Tel. +43(0)1/58801-0 <http://www.tuwien.ac.at>

Erklärung zur Verfassung der Arbeit

Patrick Zwickl, Gröhrmühlgasse 36E, 2700 Wiener Neustadt

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit einschließlich Tabellen, Karten und Abbildungen, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Dezember 2009

Patrick Zwickl

Danksagung

Im Zuge meines Studium wurde ich von vielen Menschen, als Kommilitonen, Lehrende, Betreuer, Freunde, Familie etc., begleitet. Dabei war es mir möglich Unterstützungen auf verschiedensten Ebenen zu erhalten oder in einer guten Zusammenarbeit respektable Ergebnisse zu erzielen. Aufgrund der größeren Zahl an Begleitern, möchte ich mich in der Danksagung im Besonderen auf jene Unterstützer beschränken, die in einem überdurchschnittlichen Zusammenhang mit dieser Diplomarbeit stehen.

Ich möchte mich sehr herzlich für die engagierte Hilfe, Leitung und Supervision bei meinen Betreuern O.Univ.-Prof. Mag. DI Dr. Gerti Kappel und Univ.-Ass. Mag. Dr. Manuel Wimmer bedanken. Darüber hinaus möchte ich mich einen besonderen Dank für die fachliche Unterstützung an DI (FH) Johannes Schönböck und DI Angelika Kusel richten. Es würde mich daher freuen, sollte sich auch in Zukunft die eine oder andere Möglichkeit der Zusammenarbeit (Supervision) bieten.

Aufgrund der aufgewandten Unterstützung und der reichlich zuteil gewordenen Geduld möchte ich mich bei meiner Familie, aber im Besonderen bei meiner Freundin, Jacqueline Lonsky, für ihre große Hilfe durch Korrekturlesen der schriftlichen Ausarbeitung bedanken.

Neben all diesen klassischen Danksagungen, erfreue ich mich besonders über jede Zeile, die von jenen Menschen gelesen werden/wurden, die dies nicht aus (moralischer) Pflicht, sondern aus Interesse an dem Thema und der geistigen Auseinandersetzung mit diesem Gebiet machen/machten. Dementsprechend möchte ich großen Dank an jene für Ihr fachliches Interesse richten.

Abstract

Model transformations (MT) play a key role in the Model Driven Engineering (MDE) paradigm, leading to the standardization of the Query/View/Transformation (QVT) model transformation language by the Object Management Group (OMG). Until now, however, this language did not attract the same interest as the Unified Modeling Language (UML), because of the lack of adequate debugging facilities which are necessary regarding the following three problem areas: First, declarative languages like QVT Relations (QVT-R) hides the operational semantics of transformations. Only the information provided by the interpreter, as well as the tendered inputs and returned outputs are available for tracking the progress of transformations. Furthermore, the ordering of transformation application is hidden by the MT engines providing only a black-boxes view to the users. This can lead to the problem of impedance mismatches between design and runtime. These characteristics of QVT-R are assets for developing, but are handicaps for debugging. Second, QVT-R code is specified on higher abstraction level than its execution and state-of-the-art debugging. This deteriorates the ability to deduce causes from produced results. Third, the information content responsible for operating MTs is spread over several artifacts including the input model, a resulting target model and the QVT-R code. As a consequence, the reasons for a particular outcome are hard to be derived from the involved artifacts. This severely harms the ease of debugging.

Therefore, this master thesis tackles the mentioned problems by visualizing QVT-R as Transformations Nets, using the MT framework “Transformations On Petri Nets In Color” (TROPIC) based on Colored Petri Nets (CPN). This can be seen as explicit definition of operational semantics on a high abstraction level providing a white-box view for debugging QVT-R. This thesis proposes a procedure model formulated in a conceptual approach and in a prototypic implementation striving for bridging the existing gap between these two different paradigms by mapping the concepts of QVT Relations to such nets. In this thesis three particular contributions are provided: (i) a solution approach for unidirectional mappings producing target models from an existing source model, (ii) the support for model inheritance, (iii) and synchronization approaches for timely and version-based incremental changes.

Kurzfassung

Modelltransformationen (MT) übernehmen eine Schlüsselrolle im Model Driven Engineering (MDE) Paradigma, welche zur Standardisierung der Modelltransformationssprache Query / View / Transformation (QVT) von der Object Management Group (OMG) führte. Allerdings konnte diese Sprache bislang nicht das gleiche Interesse wie die Unified Modeling Language (UML) wecken, da ein permanenter Mangel an adequaten Debugging Mechanismen besteht. Folgende drei Problembereiche wurden dabei identifiziert: Erstens offerieren deklarative Sprachen wie QVT Relations (QVT-R) keine operative Sicht des Transformationsprozesses. Nur die Informationen, die von den Interpretern zur Verfügung gestellt werden, sowie die vorhandenen Ein- und Ausgabedaten, können für die Beobachtung der Transformationsumsetzung genutzt werden. Sogar die Reihenfolge der Transformationsausführung wird durch den MT engine als Black-box-System verschleiert. Dies kann zum Problem der Diskrepanz zwischen Design- und Laufzeit führen. Zweitens verfügt der QVT-R Code über ein höheres Abstraktionslevel als die Ausführung und das Debugging des Codes. Dies beeinträchtigt die Möglichkeit von Schlussfolgerungen aus produzierten Ergebnissen. Drittens sind die Informationen der Durchführung der Modelltransformationen über mehrere Artefakte – einschließlich des Source-Modells, des resultierenden Ziel-Modells und des umfassenden QVT-R Codes – zerstreut. Daraus folgend sind die Gründe für ein bestimmtes Ergebnis für den Benutzer nicht nachvollziehbar.

Zur Lösung der genannten Probleme wird in dieser Diplomarbeit die Visualisierung von QVT-R in Transformationsnetzen mittels des MT Frameworks “Transformation On Petri Nets In Color” (TROPIC), das auf Colored Petri Nets (CPN) aufbaut, erörtert. Diese kann als explizite Definition der operationalen Semantik für die Fehlersuche auf hoch abstraktem Level interpretiert werden. Die vorliegende Arbeit formuliert einen konzeptionellen Ansatz mit protoypischer Umsetzung zur Überwindung der bestehenden Kluft zwischen den verwendeten Paradigmen. Dies erfolgt durch die Abbildung der QVT-R Konzepte auf TROPIC. Konkret werden drei Kontributionen präsentiert: (i) ein Lösungsansatz für unidirektionale Transformationen welche ein Zielmodell aus einem bestehenden Quellmodell erstellen, (ii) die Unterstützung von Modelvererbung, und (iii) Synchronisationsansätze für zeitliche und versionsbedingte inkrementelle Änderungen.

Contents

| | |
|--|-----------|
| 1. Introduction | 1 |
| 1.1. Motivation | 1 |
| 1.2. Problem Statement | 1 |
| 1.3. Solution | 2 |
| 1.4. Structure of the Thesis | 3 |
| 2. Model Transformation Fundamentals | 5 |
| 2.1. Model Driven Engineering | 5 |
| 2.2. Model Transformation | 6 |
| 3. Model Transformation Technologies | 9 |
| 3.1. Query/View/Transformations (QVT) | 9 |
| 3.1.1. QVT Relations (QVT-R) | 10 |
| 3.2. Transformation On Petri Nets In Color (TROPIC) | 14 |
| 3.2.1. Core concepts of TROPIC | 15 |
| 3.3. Comparison of QVT Relations and TROPIC at a glance | 18 |
| 4. Transforming QVT Relations to TROPIC | 21 |
| 4.1. Overview | 21 |
| 4.1.1. Execution Semantics of QVT-R | 23 |
| 4.2. Types | 26 |
| 4.2.1. Domains | 28 |
| 4.2.2. Primitive Domains | 29 |
| 4.2.3. Primitive types | 30 |
| 4.2.4. Complex types | 30 |
| 4.3. Relationships | 32 |
| 4.4. Dependencies | 34 |
| 4.4.1. Where- and When-Clauses | 34 |
| 4.4.2. Hierarchical Data Passing | 35 |
| 4.5. Finding Correspondence of Source and Target Objects | 37 |
| 4.5.1. Heavyweight Approach | 37 |
| 4.5.2. Lightweight Approach | 39 |
| 4.6. Realization of Correspondences | 41 |
| 4.6.1. Transition “Owners” | 41 |
| 4.6.2. Relationships | 42 |
| 4.6.3. Referred Relations | 44 |
| 4.7. Complexity and Effort | 46 |
| 5. Supporting Advanced Features of QVT Relations | 51 |
| 5.1. Inheritance | 51 |

| | | |
|-----------|---|------------|
| 5.2. | Incremental Changes | 52 |
| 5.2.1. | Introduction | 52 |
| 5.2.2. | Model Synchronization | 52 |
| 5.3. | Summary | 55 |
| 6. | Realization | 57 |
| 6.1. | Environments | 57 |
| 6.1.1. | Eclipse and its Plugins | 57 |
| 6.1.2. | QVT-R Parser | 58 |
| 6.2. | Implementation Specifics | 59 |
| 6.2.1. | Data Types | 59 |
| 6.2.2. | Information Encapsulation | 60 |
| 6.2.3. | Class Structure | 61 |
| 6.3. | Execution Process | 64 |
| 6.3.1. | Course of Execution | 64 |
| 6.3.2. | Managing Flow Information | 71 |
| 6.3.3. | Managing colours in TROPIC | 72 |
| 6.3.4. | Adding Model Information | 72 |
| 6.3.5. | Element Arrangement | 75 |
| 6.4. | Summary | 76 |
| 7. | Evaluation | 79 |
| 7.1. | Capability of Erroneous Code Recognition | 79 |
| 7.1.1. | Pitfall Detection | 79 |
| 7.2. | Visualization | 81 |
| 7.2.1. | Performance | 83 |
| 7.2.2. | Unavailable Language Elements | 84 |
| 7.2.3. | Effort Increase Analysis for Transformation Net Creations | 84 |
| 7.3. | Model Synchronization | 86 |
| 7.4. | Summary | 87 |
| 8. | Related Work | 89 |
| 8.1. | Testing | 89 |
| 8.1.1. | Trial & Error | 89 |
| 8.1.2. | Test-Driven-Development | 90 |
| 8.2. | Debugging | 91 |
| 8.2.1. | Model-based Debugging | 92 |
| 8.2.2. | QVT-R Debuggers | 92 |
| 8.2.3. | Forensic Debugging Techniques | 94 |
| 8.2.4. | Backwards Debugging | 94 |
| 8.3. | Verification | 95 |
| 8.3.1. | Verification by Translation | 95 |
| 8.3.2. | Verification of Graph Transformations | 96 |
| 8.3.3. | Verification based on CPN | 96 |
| 8.4. | Summary | 99 |
| 9. | Conclusion | 101 |
| 9.1. | Summary | 101 |

| | |
|----------------------------|------------|
| 9.2. Future Work | 101 |
| A. Appendix | 103 |
| Bibliography | 105 |

1. Introduction

1.1. Motivation

Over the last years Model Driven Engineering (MDE) experienced a strong impetus, placing models as first class artifacts throughout the software lifecycle. Although models are used in software development since several decades up to now [7], documentation was their main purpose only. In contrast to that, models in MDE not only serve for documentation, but the main promise of MDE is to raise the level of abstraction from technology and platform-specific concepts to platform-independent and computation-independent modeling [3].

This success exceptionally qualifies and advocates models – such as models from the Unified Modeling Language (UML) standard – to be used for architectural issues to create a common understanding. Typically several model types are used which is apparently related to the complexity that modern software development has to face. Therefore, it is often necessary to use a set of different model types for representing a particular environment that is directly related to a single domain specific model (or to all other models). Hence, automatization or even synchronization mechanisms for model transformations (MT) from certain source models to desired target models are needed as they can improve the consistency of models and can reduce the effort.

To fulfill this promise, the availability of appropriate model transformation languages is the crucial factor, since transformation languages are as important for MDE as compilers are for high-level programming languages. Transformation scenarios can be divided into vertical model transformations and horizontal model transformations. Vertical model transformations lower the level of abstraction, e.g., generation code from UML class diagrams, whereas horizontal model transformations transform models between two different representations on the same level of abstraction, which is the focus of the rest of this thesis, e.g., a UML class model is transformed to an entity relationship diagram. There already exist several model transformation languages (MTL) that provide the transformation from one model (conforming to a meta model) to another model (conforming to another meta model), e.g., the Query/View/Transformation (QVT) standard [22]. Typically a rich set of transformation rules – defined by the developer – is necessary to determine which structure(s) of the source model correspond to which structure(s) of the target model.

1.2. Problem Statement

In this thesis the declarative model transformation language QVT Relations (QVT-R) is used which has been standardized by the Object Management Group.

1. Introduction

Although model transformations are highly necessary in the context of model driven approaches, not even the QVT standard was able to attract sufficient interest until now. A main factor for this weak adoption results from the lack of adequate debugging facilities. Such facilities are highly necessary according to the three particular identified problem areas:

First, declarative languages like QVT-R cannot make the operational semantics of transformation specifications explicit to the transformation designer. Only the information provided by the interpreter, as well as the tendered inputs and returned outputs are available for tracking the progress of transformations. Even the ordering of rule application is hidden by the MT execution engine which is provided as a black-box to the users. This can lead to the problem of impedance mismatches between design time and runtime. This characteristic of QVT-R is a valuable asset for developers as they need not specify these details, but for debugging this can be seen as severe handicap. Second, QVT-R code is specified on a high-level of abstraction whereas its execution and state-of-the-art debugging mechanisms are positioned on significant lower level of abstraction. This deteriorates the ability to deduce causes from produced results. Third, the information content responsible for operating MTs is spread over several artifacts including the input model, a resulting target model and a comprehensive QVT-R code. As a consequence, the reasons for a particular outcome are hard to be derived from the involved artifacts. This severely harms the ease of debugging.

These three problem areas are, moreover, influenced by the complexity of typical model transformation rules and the unclear semantics of some QVT-R language elements – e.g., “check” and “enforce”.

1.3. Solution

This master thesis concentrates on several areas of improving the ease and quality of debugging. It discusses the possibilities of visualizing QVT-R in “Transformations On Petri Nets In Color” (TROPIC) – which uses Transformation Nets based on Colored Petri Nets (CPN). This can be seen as an explicit definition of the operational semantics providing a white-box view for debugging QVT-R. This thesis proposes a conceptual approach that is transposed into a prototypical implementation. It strives for bridging the existing gap between the desired operational semantics for debugging and the declarative language elements (as used in QVT-R) by mapping the concepts of QVT-R to TROPIC. TROPIC can provide a white-box-view – as paths from source to target representation by using Arcs and Transitions – of the code on a high-level of abstraction by visualizing the involved metamodel structures (as Places) and the concrete model information (as Tokens). It, therefore, allows the analyzation of the before scattered content in one single visualization. In the first step the focus lies on the detection of solutions for unidirectional transformation mappings. This is deepened in the second step by providing support for advanced features such as inheritance used in input or output models. The third will concentrate on the fundamental problem of incremental changes by proposing synchronization mechanism theorems and providing a simple prototypic implementation. Incremental changes are modifications of a previously created model whereas synchronization approaches

can assist to hold related models consistent in the case of modifications.

The thesis finalizes by an evaluation of how the proposed concepts are appropriate for the specified context and which challenges and possible exploitations of the TROPIC runtime model, e.g., complex Object Constraint Language (OCL) statements [21], remain for future research.

1.4. Structure of the Thesis

The rest of the thesis is structured as follows. Section 2 introduces the fundamentals of model transformations and debugging facilities. This thesis proceeds with the used technologies and standards. The major component of this theses is the practical part according to the focus on a set of scientific solution concepts and considerations. This part starts with an introduction of the conceptual approach of this paper in section 4 leading to advanced problems in section 5. The actual realization is presented in section 6 and is evaluated in chapter 7. This thesis finalizes with the presentation of related works and approaches (section 8), and a summarization of results (9).

1. *Introduction*

2. Model Transformation Fundamentals

There exist some fundamental areas in the field of modeling and metamodeling that are introduced in this section. The differentiation between abstraction levels is essential for the usage of model-driven approaches. How such model-driven approaches can be integrated in modern software development is introduced as well. As a result of the spreading of model-based approaches for software development, model-to-model transformations are necessary and, therefore, introduced in this section.

2.1. Model Driven Engineering

Model Driven Engineering (MDE) places models as first class artifacts throughout the software lifecycle. A particular approach for MDE is presented by the Object Management Group with their Model Driven Architecture (MDA) approach [23]. In MDA this means the usage of a Platform-Independent-Model (PIM) as abstract architectural representation that is transformed to more concrete Platform-Specific-Models (PSM) or even source code. The usage of several models on different levels of abstraction for different purposes and the intended generation of the more specific models from a more general representation clearly states the importance of model transformations in the context of MDE.

Furthermore, it is necessary for the understandability and usage of MDE to specify the abstraction levels of models. The execution of a program is placed on level M0. From this execution a model (M1) can be built abstracting “real world” problems to make it ascertainable for humans. But normally models are not built by just drawing some lines and boxes, they actually follow grammars consisting of rules how to build this type of model. For example a UML class diagram has restrictions according to types that can be used and in which relations they have to stand (and many more rules that have to be followed). This structure is defined on meta level (M2) by a metamodel. One abstraction level higher (M3), it reaches the abstraction level of metamodeling languages like the recursively defined Meta Object Facility (MOF). At this level the boundaries for building metamodels and Modeling languages respectively are set.

A simple overview of the abstract levels provides Figure 2.1. The left branch of the diagram shows the source model **Ma** (input) that conforms to its metamodel **MMa**. On the right branch the target model **Mb** (output) is placed that conforms to the metamodel **MMb**.

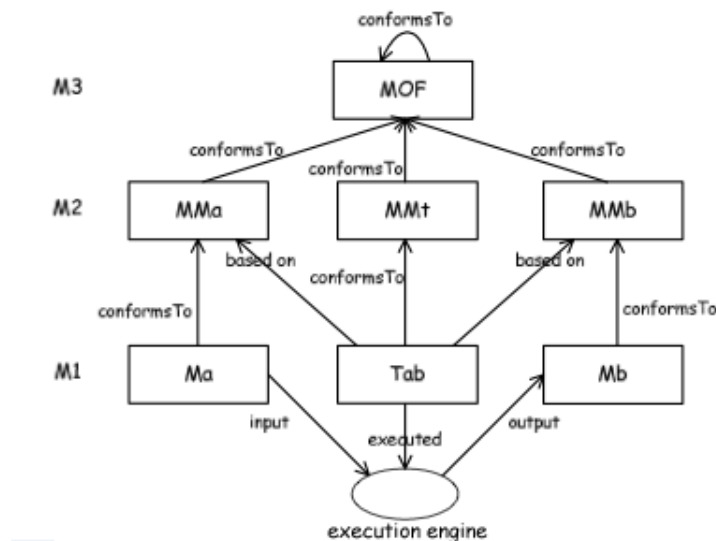


Figure 2.1.: Abstraction levels in the context of model transformation [13]

2.2. Model Transformation

Transformations languages like QVT transform models (M1) to other models. To be able to transform all models conforming to a metamodel (M2) to a model conforming to another metamodel, the metamodels are used for defining the particular transformation rules. Metamodels are used as they provide the knowledge of which elements can occur in a particular model. In Figure 2.1 the central branch is the transformation combining source and target models conforming to a metamodel `MMt`. `Tab` represents the actual transformation program. All metamodels (`MMa`, `MMb`, `MMt`) again conform to a more abstract model – the MOF metamodel. The **execution engine** consumes some input and produces some output data and interacts with the transformation code of the program.

In MDA the transformation from PIM to PSM is a vertical transformation. Additionally, there are horizontal transformations that do not change the abstraction level, but transform the model to fulfill the needs of another language (according to domains and special needs) or perspective. In the context of this thesis the focus is on horizontal model to model transformations.

According to Juan de Lara and Esther Guera [7] model transformations can be divided into: “Inter-Formalism (also known as exogenous)”, specifying a transformation between different source and target metamodels, and “Intra-formalism (also known as endogenous)”, specifying a transformation where source and target metamodels are equal.

The actual transformation from a source to a target model is done using a transformation language like QVT-R or Atlas Transformation Language (ATL). There exist some requirements and some desired features for transformation languages which are declared in the thesis of Thomas Reiter [25]).

Furthermore, according to Juan de Lara et al. [7] transformation languages can be categorized by “three orthogonal characteristics in the transformation language”: (i) visual or textual, (ii) imperative, declarative, (iii) formal or semi-formal.

(i) Most transformation and programming languages can be considered as textual representations. For example Java [7] and ATL are obviously textual as they use text in lieu of graphical representations such as boxes, or graphs. In contrast to graph transformations can be considered to be visual.

(ii) Declarative languages are comfortable to use and reduce the lines of code necessary. If all elements in all facets from a source model can be fully mapped using declarations, such usages are to be favoured according to its simpleness. The model transformation language engine is then responsible for changing these declarations in clear imperative commands in the back-end. This ensures quality and performance and reduces avoidable mistakes such as mixing commands, variables or names, or the improper ordering of rules, because such facets are automatically and implicitly handled by the transformation engines.

Imperative languages are very similar to normal programming languages and, therefore, very flexible. Writing many commands for one actual transformation rule seems somehow unpurified and makes it hard to find errors. Therefore, it is considered good practice only using imperative languages, if declarative mappings cannot be used.

For the scenario of complex transformation rules in the same transformation process, hybrid languages can be used trying to combine the benefits of both parts. Hybrid languages like the Atlas Transformation Language (ATL) allow declarative statement blocks as well as imperative statement blocks.

(iii) Formal languages are defined by a clear alphabet used to form a grammar that specifies the combination of elements from the alphabet. A full formal language does leave any gap for interpretation. This is not the case for typical programming – like Java – and model transformation languages. For example Java and QVT-R (and similar languages) are semi-formal as they provide a formal alphabet and grammar, but leave gaps for interpretability.

2. *Model Transformation Fundamentals*

3. Model Transformation Technologies

To be able to run model transformations, a set of technologies and standards are necessary to be used. In the approach presented in this thesis, the model transformation language QVT Relations is used and visualized in TROPIC Nets. Both QVT Relations and TROPIC are presented isolated in this section as preparation for combining them later on. Finally, QVT Relations and TROPIC are compared with each other to identify possible ways of transforming elements of QVT Relations to TROPIC in following sections.

3.1. Query/View/Transformations (QVT)

This thesis uses the Object Management Group (OMG) standard Query/View/-Transformations (QVT) [22] ; a part of the Meta/Object/Facility (MOF) for model transformations. QVT is a transformation language that can be split into the three parts (1) Core, (2) Relations and (3) Operational transformations, shown in Figure 3.1). The Core and Relations component build up the declarative part of the language. Imperative transformations are done using the Operational component which is not used in this thesis. There also exists the possibility of transforming code from the Relations to the Core language to specify the operational semantics of QVT-R.

Furthermore, codes written in another programming language offering a “MOF binding” [22] can be plugged-in as black-box systems. They have to be considered as black-box, because their actual implementation is not accessible or visible from the QVT code.

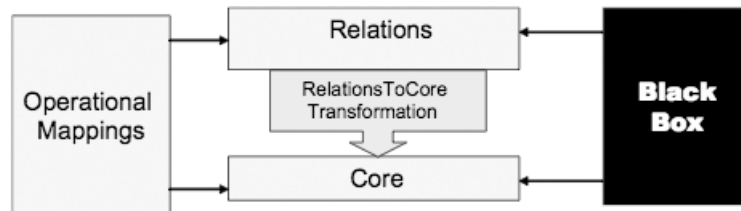


Figure 3.1.: QVT components and interactions [22]

The possibility of using different transformation approaches (declarative, operational) is reflected in the component-oriented architecture of this MTL. Additionally, QVT is not a standalone language. For example OCL expressions are used for the definition of QVT Relations. QVT is a very versatile language that allows many different queries according to its powerful definition. Besides this

integration of other languages QVT also relies on other standards, e.g., the Meta Object Facility (MOF).

Although the QVT specification has already reached a professional and utilizable level, there exist some open issues. For example it is unclear how the ordering of rules should be accomplished by the engines. Such issues do not directly reflect the particular elements of the MTL, but their conversion to real systems.

3.1.1. QVT Relations (QVT-R)

QVT-Relations can be used for uni- and bidirectional transformations. After creating a certain QVT-R code, it can be executed by QVT-R engines using a source model, an source metamodel and an target metamodel. There is even the possibility of translating QVT-R transformations to QVT-C to execute them on QVT-C engines. However, the QVT-R and QVT-C engines are rare.

In the following section technical details of the QVT-R language are presented.

Language introduction

In QVT Relations “a transformation between candidate models is specified as a set of relations that must hold for the transformation to be successful” [22].

Transformations have to be invoked in a specific direction. This means that it has to be defined which model with which metamodel acts as source model and which model acts as target model. Typically QVT-R is used to read model elements from the source model, to check if some conditions are fulfilled and to finally produce the target model. However, QVT-R can also be used to compare source and target model with each other.

Available QVT Relations (QVT-R) engines/tools supporting many features of this language – according an article of Ivan Kurtev [14] - are IK++ medini QVT, Eclipse M2M, Relations2ATLVM or Tata Consultancy ModelMorf. In the context of this thesis there will be a strong focus on the usage of medini QVT¹, because it has been widely used at the moment of writing and is able to handle many features of the QVT-R language.

The term “mapping” in the context of QVT (QVT-Relations) refers to the rules for transformations. One rule, therefore, is a mapping. In the context of this thesis the term “mapping” is, therefore, used synonymously.

Relations and Domains

A Relation is a definition holding concrete transformations and pre- and post-conditions that are directly related to this particular set of transformation statements. Such transformations can be compared to constraints that need to be fulfilled to allow the execution for a particular object. A Relation is defined by at least two Domains and optional When-/Where-Clauses.

“A domain is a distinguished typed variable that can be matched in a model of a given model type. ” [22]. A domain follows a pattern that represents a number of included types.

¹<http://www.ikv.de/> - last accessed: November 27 2009

The example listing on the left side of Figure 3.2 shows the definition of a Relation to transform a **class** to a **table** – conforming to the source and target metamodel depicted on the right side of Figure 3.2 – including Domains as well as **Where** and **When** Clauses. The Domain **c:Class** holds several variables – the pattern of this Domain – representing specific constraints that need to hold. For example **kind='Persistent'** means that the variable **kind** has to have the type **'Persistent'**. On the other side **name = cn** in the Domains **c:Class** and **t:Table** means that the names have to be the same – **cn** is a placeholder for this constraint.

Where and When

The Where-Clause specifies a post condition for a Relation. This means that the stated Relation has to be called, if the calling Relation itself has been called. This has to be interpreted as explicit call of the stated other Relation with the specified parameters, i.e. variables identifying the actual Domain. In contrast When-Clauses specify preconditions that have to hold before a Relation can be executed. This does not mean that the When-Clause forces the referred Relation to be called, but it waits for its execution. If the referred Relations is never called from somewhere else, the Relations holding this When-Clause cannot be executed.

See the listing in Figure 3.2 for the usage of When- and Where-Clauses within a Transition. The When-Clauses refers to the Relation **PackageToSchema** and involves the variables **p** and **s**. The Where-Clause explicitly calls the Relation **AttributeToColumn** with the date of the variables **c** and **t**.

Top level

The transformations discussed so far were non-top level Relations. These Relations have to be invoked directly or indirectly (transitively using where clauses). Beyond that there also exists the type of top level Relations. They hold the keyword **top**. All top level Relations have to be executed. Non-top level need not to be executed, if they are not invoked. See Listing 3.1 for the specification of the different types of Relations. The top level Relation **ClassToTable** invokes the non-top level Relation **AttributeToColumn** in its Where Clause.

Listing 3.1: A QVT transformation holding different types of Relations [22]

```

1
2 transformation umlRdbms (uml : SimpleUML, rdbms :
   SimpleRDBMS) {
3   top relation PackageToSchema {...}
4   top relation ClassToTable {
5     where {
6       AttributeToColumn(c, t);
7     }
8   }
9   relation AttributeToColumn {...}
10 }
```

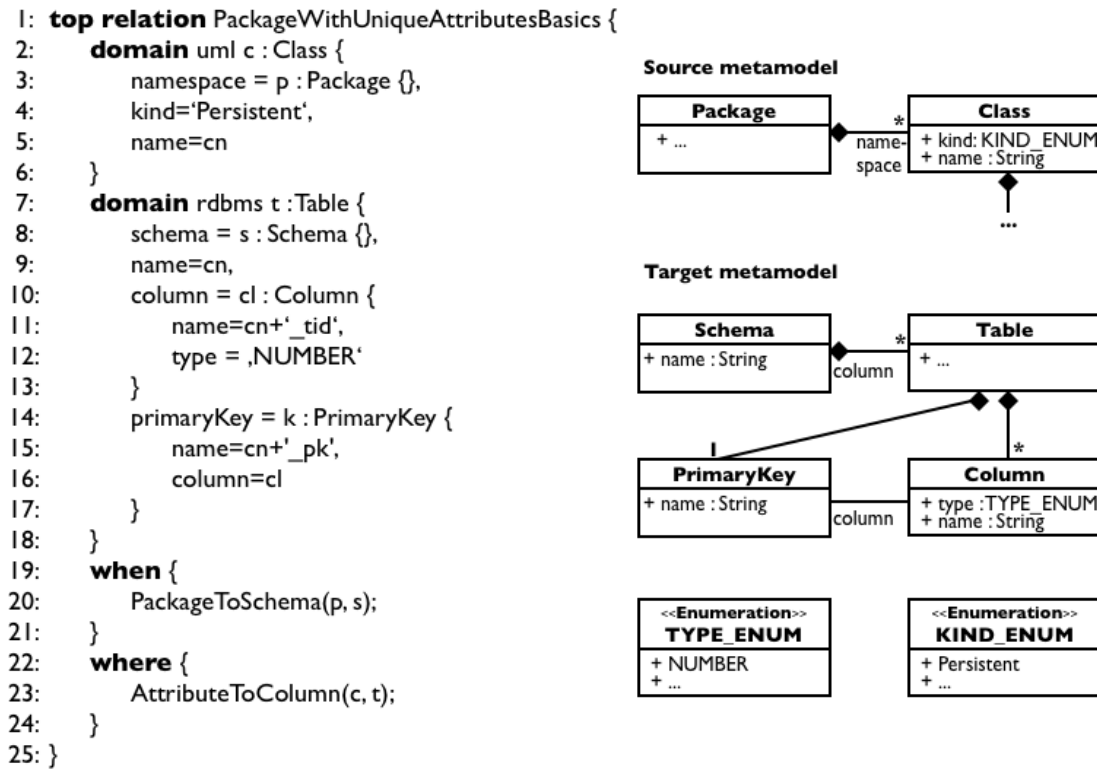


Figure 3.2.: Left side: QVT Relation with Domains and When- and Where-Clauses. Right side: Related source and target metamodel. [22]

Check and Enforce The Relations language makes use of the Check (Checking) and Enforce (Enforcement) elements, as depicted in Listing 3.1.

- **Checking:** The checking is a “weaker” mode than the enforcement mode. It simply checks, if the constraints of models are fulfilled in the context of this transformation, and reports errors if any constraints are violated. However, this mode implies that nothing is “produced” in the sense of model transformations.
- **Enforcement:** The use of enforcement mode defines the transformation direction. This means “the selection of one of the candidate models as the target model. The execution of transformation proceeds by, first checking the constraints, and secondly attempting to make all the violated constraints hold by modifying only the target model and the trace model.” [22]

Listing 3.2: Checkable and enforceable Domains [22]

```

12 relation PackageToSchema
13   /*map each package to a schema */
14   {
15     checkonly domain uml p:Package {
16       name=pn
17     }
18     enforce domain rdbms s:Schema {

```

Metamodel

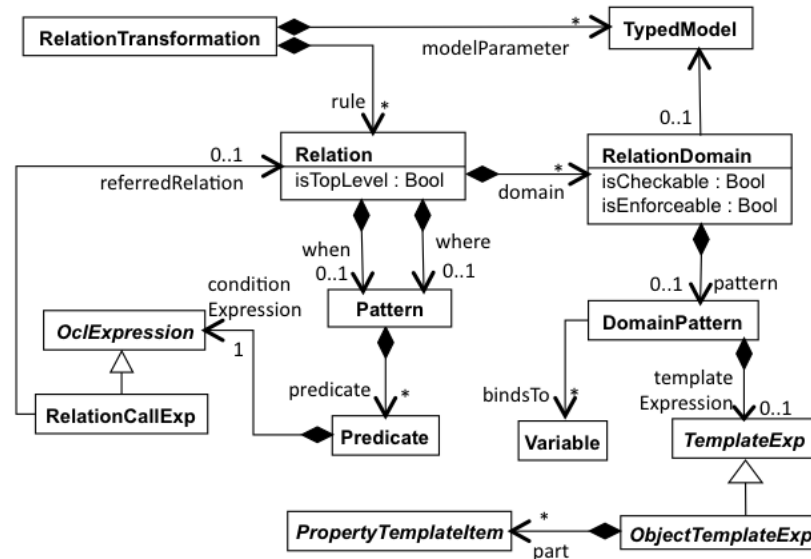


Figure 3.3.: The basic meta model of QVT Relations [29]

of `DomainPattern` – classically one. In this pattern the actual mappings take place. At least one complex type (`ObjectTemplateExp`) is contained that can hold several other types. `PropertyTemplateItem` objects are used to match for attributes in the Domain pattern. Primitive (several types) and complex types refer to subtypes of the `OclExpression` which represents a detail left out in the shown metamodel.

QVT-R can be transformed to QVT Core to represent the formal semantics behind the statements such as Relations.

3.2. Transformation On Petri Nets In Color (TROPIC)

“Transformations On Petri Nets In Color” (TROPIC) [25] is a project presented in the thesis of Thomas Reiter - Johannes Kepler Universität Linz at the Institute of Bioinformatics - to propose “a dedicated transformation execution model based on Colored Petri Nets, which allows to combine the statefulness of imperative approaches as well as raised level of abstraction from declarative approaches” [25]. This master thesis uses TROPIC to improve the ease of debugging of “classical” transformation languages – respectively QVT-R. In particular, for every meta-model element corresponding Places in TROPIC are derived, whereby a Place is created for every class, every attribute and every reference. Model elements are represented by Tokens which are put into the according Places. Finally, the actual transformation logic is represented by Transitions. The existence of certain model elements (i.e., Tokens) allows Transitions to fire and thus stream these Tokens from source places to target Places representing instances of the target metamodel to be created (see Figure 3.4).

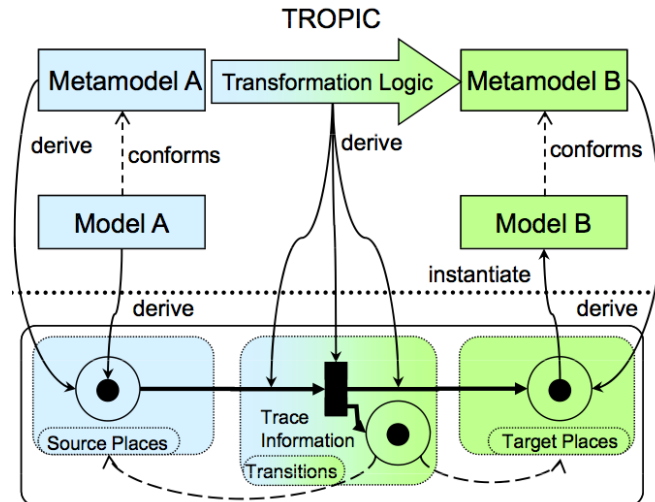


Figure 3.4.: The involved artifacts in TROPIC [30]

The stepwise firing of the Transitions makes the operational semantics of the transformation logic explicit and thereby enables simulation. The ability to combine all the artifacts involved, i.e., metamodels, models, as well as the ac-

tual transformation logic, into a single representation makes the formalism especially suited for gaining an understanding of the intricacies of a specific model transformation. Moreover, TROPIC form a runtime model, serving as an execution engine for diverse model transformation languages, e.g., QVT-R. Therefore, TROPIC exceptionally qualifies for trying to uncover the elements hidden by the QVT-R declarations as black-box statements.

In this section the main features of TROPIC nets are explained that together with QVT built up the basis for the later on proposed method to translate QVT-R code to TROPIC in order to debug model transformations.

3.2.1. Core concepts of TROPIC

TROPIC uses a different perspective on transformation problems as MTLs like QVT-R. Since TROPIC is based on Colored Petri Nets, the main parts of the TROPIC metamodel (see Figure 3.7) consists of **Places**, **Tokens** and **Transitions**.

The **Net** is the root element holding all other TROPIC items. There exist one- and two-coloured – represented by a doubled border – **Places** representing (concrete or abstract) classes (see Figure 3.5). The objects (the instances of these **classes**) are represented by **Tokens**. One-coloured **Tokens** can only be added to one-coloured **Places** and two-coloured **Tokens** to two-coloured **Places**. Places itself are, therefore, the holders of the state in TROPIC. The colours of **Tokens** are used to differentiate between different object instances they represent. **TropicUnit** can be used for clustering other TROPIC units together in meaningful groups.

Models in this thesis - including the TROPIC models – are based on Ecore. Ecore is an implementation for a subset of MOF and allows the usage of so called **EClasses**, **EAttribute** and **EReference**. **EClasses** are comparable with classes in UML class diagrams holding relationships – **EReference** – and **attributes** – **EAttribute**. Such a mapping of MOF model concepts to TROPIC can be seen in Figure 3.5 – in this Figure **EClass**, **EAttribute** and **EReference** are named as **Class**, **Attribute** and **Reference**.













| | MOF Concept | | TROPIC Concept | |
|--------------------|-------------------------------|---|--|---|
| Metamodel Elements | Class |  | OneColoredPlace |  |
| | Attribute |  | TwoColoredPlace |  |
| | Reference |  | TwoColoredPlace |  |
| Model Elements | Object (Instance of Class) |  | OneColoredToken contained in a OneColoredPlace |  |
| | Value (Instance of Attribute) |  | TwoColoredToken contained in a TwoColoredPlace |  |
| | Link (Instance of Reference) |  | TwoColoredToken contained in a TwoColoredPlace |  |

Figure 3.5.: Comparing MOF and TROPIC elements on model and metamodel level [15]

3. Model Transformation Technologies

Transitions consist of input placeholders (LHS of the **Transition**) representing the preconditions of a certain transformation, whereas output placeholders (RHS of the transition) depict its postcondition. Those placeholders are expressed by the classes **InPlacement** (LHS) and **OutPlacement** (RHS) in the metamodel as shown in Figure 3.7. Every **Placement** is connected to a source or target **Place** using **Arcs**, whereby incoming and outgoing **Arcs** are represented by the classes **PTArc** and **TPArc**, respectively. To express these pre- and postconditions, so-called **MetaTokens** (see class **MetaToken** in the metamodel in Figure 3.7) are used, prescribing a certain **Token** configuration by means of colour patterns which can be used in two different ways, either as **Query Token** (LHS) or as **Production Token** (RHS), as shown in Figure 3.6. The **Transitions**, therefore, provide the construction of a transformation flow by using certain colour patterns. **Query Tokens** and **Production Tokens** using same colours imply a correspondence of LHS and RHS elements. Colours only used on one side represent unique or new colours (if used as **Production Tokens**).

Different colours – although they are not forced to be used – are necessary for aggregating and differentiation between different values of **Tokens**. However, two-coloured **MetaTokens** can be used for aggregating values from two one-coloured objects. This is achieved by placing two one-coloured **MetaTokens** in **InPlacements** of the same **Transition**. Furthermore, a two-coloured **MetaToken** is placed in an **OutPlacement** of the same **Transition** holding the same colours as both **MetaTokens** in the **InPlacements**. Naturally, this procedure can be used to split up values or to newly address values, but it clearly states that colours are the only method that can be used in TROPIC to differentiate between different values.

The importance of colouring **MetaTokens** is also describable by using **Tokens** as conditions to fire **Transitions**. For example it can be defined that the **Token** “isPersistent=true” is a necessary condition for a **Transition** that transform the **Token** “Class” to a “Table” (**Place**). Hence, many **Tokens** can be necessary to fire a **Transition** that produces just a single output **Token**. To be able to produce comprehensive transformations there is the possibility to create sequences of **Transitions**. A **Transition** delivering a **Token** to a **Place** need not be the first nor the last **Transition** in a row. The receiving **Place** can again have an outgoing **Arc** to another **Transition** or even **Arcs** to a handful of **Transitions**. This also implies the possibility for a **Place** to have several **Arcs** from **Transitions** filling it with **Tokens**.

Furthermore, **Transitions** can hold **InPlacements** that are hungry or not hungry (standard). Hungry means that by firing a **Transition** the used inputs are removed from its original **Place**. The overall principle of passing **Tokens** on (or even modifying them) stays the same.

A colour that is not used for **MetaTokens** on both sides like the blue **MetaToken** in Figure 3.6, represents a newly created colour. This represents one-sided conditions. Colours used on both sides represent value correspondences. Colours reused on the same side (LHS or RHS) represent model dependencies such as from a **class** to its **attributes**. This accomplished by using two-coloured **MetaTokens**.

Furthermore, configurations can be used that describe the **Transition** behaviour that should be applied when firing it. Such configurations can be used

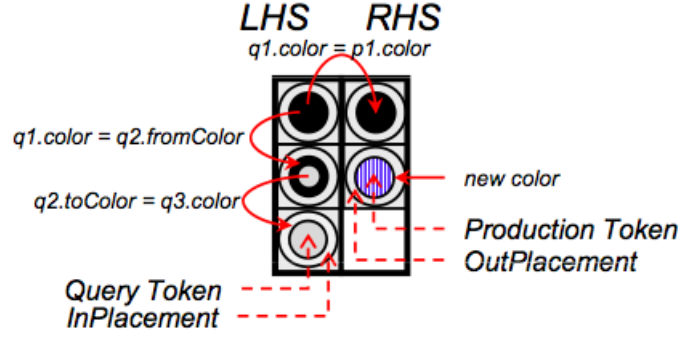


Figure 3.6.: Colour binding example of TROPIC [31]

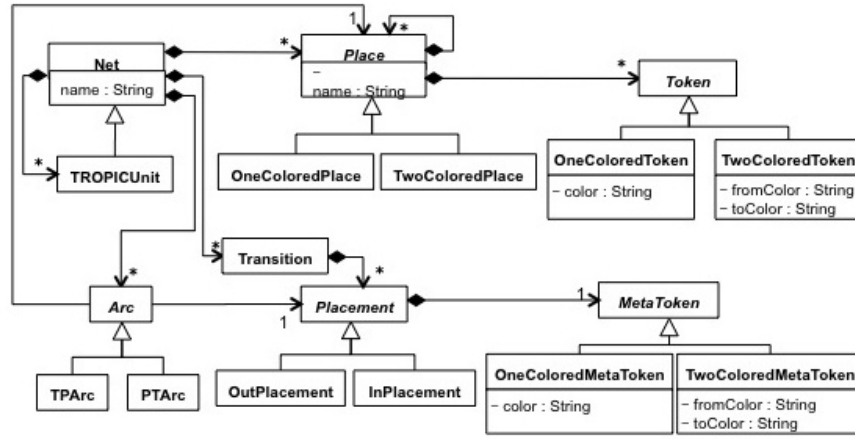


Figure 3.7.: Metamodel of TROPIC [29]

by defining functional constraints. A configuration is directly applied to the **Transition** and is not automatically used for predecessors or successors.

Some typical **Transitions** are explained in the Figure 3.8. A simple **Transition** just “waits” for an incoming **Token** and passes it in the same colour to the output **Place** (cf. Figure 3.8 (a)). **Transitions** for Two-coloured **Tokens** can also be used to invert the colours of incoming **Tokens** (cf. Figure 3.8 (b)). Furthermore, two one-coloured incoming **Tokens** can produce a two-coloured output **Token** (cf. Figure 3.8 (c)). More complex **Transitions** could use some incoming two-coloured (or even one-coloured **Tokens**) and mix the colours to new one or two coloured **Tokens** (cf. Figure 3.8 (d)). Another interesting example is the usage of horizontally aligned **Tokens** which can be compared to a logical disjunction (“or”-expression). Vertically aligned **MetaTokens** can be used that again express a logical disjunction (cf. Figure 3.8 (e)). This means that whenever a two-coloured **Token** with black (outer) and white (inner) – representing two different colours – or a **Token** with black (outer) and black (inner) – representing the same colour – is available in the **Place** related to this **Transition** it can fire and produces a **Token** with turned colours (according to the configuration of white (outer) and black (inner) for the **MetaToken** on the right side.). This example produces a two-coloured **Token** using two colours what can lead to unambiguous results. In the case of a two-coloured **Token** that is built up by just one colour (the right

alternative of the incoming **MetaTokens** is addressed) it is not clear which colour should be used. This problem can be addressed by using ColorMaps.

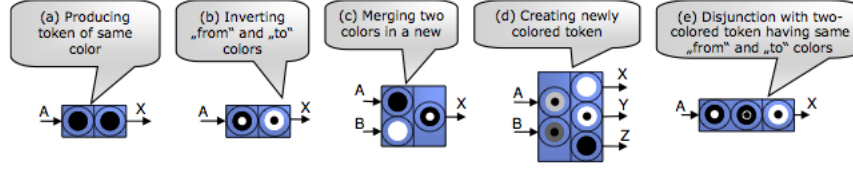


Figure 3.8.: Typical variants of Transitions [25]

So far only “positive” conditions were introduced. This means that for example a **Token** is expected to be able to fire a **Transition** from a certain **Place**. The other way round is the possibility to use “Negations” that negate the specified condition. Enabling the negation of this condition for the net (and the related formula) transfers the meaning in its opposite. E.g., a **Transition** can be fired, if a certain “Class” **Token** is unavailable for this **Transition**.

3.3. Comparison of QVT Relations and TROPIC at a glance

A simple element to element mapping from QVT-R to TROPIC is summarized in Table 3.1 and also described in [29]. The most essential element in QVT-R code is the **RelationTransformation** that holds all **Relations** responsible for the transformation execution. Such a **RelationTransformation** can be considered as the full transformation net (**Net**). As a consequence only a single **RelationTransformation** can be held by a **Net**.

In TROPIC, **Tokens** represent a certain object instance, whereas in QVT-R there are no means to explicitly represent model elements. The passing of data from source to target side is done in QVT-R by variables and in TROPIC this can be visualized with the colour of the used **MetaToken** and **Token** (for the instance itself).

The **Relation** is compared with a **TropicUnit** which represents a modularization concept for the actual transformation logic. **DomainPatterns** representing the transformation mappings, can be compared with a **Transition**. The properties included in the Domain - or rather the Pattern of the Domain - can have the type **ObjectTemplateExp** or **PropertyTemplateItem** as described in more details later on. They can be compared with a **Placement** and a **OneColoredMetaToken** or **TwoColoredMetaToken** respectively. The concept of a **When-** and **Where-**clause can be visualized using an **InPlacement** / **OutPlacement** and an **Arc** from or to a **trace place**.

How this transformation from QVT-R to TROPIC – including the usage of input models – can be accomplished in an application, is described in detail in the course of the presentation of the solution approach.

3.3. Comparison of QVT Relations and TROPIC at a glance

| QVT Relation Concept | TROPIC Concept |
|------------------------|--|
| RelationTransformation | Net |
| n.a. (Model element) | Token |
| Relation | TropicUnit |
| Execution direction | Arc |
| DomainPattern | Transition |
| ObjectTemplateExp | Placement + OneColoredMetaToken |
| PropertyTemplateItem | Placement + TwoColoredMetaToken |
| Variable | Colour of MetaToken |
| When-clause | InPlacement + PTArc from dependent trace place |
| Where-clause | Trace place + TPArc to InPlacement |

Table 3.1.: Mapping QVT Relations to TROPIC [29]

3. *Model Transformation Technologies*

4. Transforming QVT Relations to TROPIC

In this part of this thesis, an approach for graphically debugging QVT-R is presented. An approach is presented that is based on live interaction to overcome the difficulty of debugging of QVT-R code and, moreover, to improve the adoption level of QVT-R. Therefore, it is necessary to highlight why a certain result could be achieved. Furthermore, TROPIC allows inspection of the actual state of a transformation execution – which is necessary for live interaction – as it visualizes the state using coloured Tokens. The overall visualization is intuitive – as it allows the tracking of paths from a certain starting point to a certain target by following simple arcs – and allows step-wise proceeding of the process. This exceptionally qualifies TROPIC to be adopted for such a debugging approach that tries to visualize the operational semantics.

Hereinafter the presented approach is called “**Graphical Debugging of QVT**” according to the name of the thesis. To simplify its usage the abbreviation “Grade QVT” or “Grade” is used. Furthermore, the meaning of this abbreviation seems appropriate as well, because to grade QVT-R could be related to the idea of debugging in general.

4.1. Overview

In this section the conceptual approach based on the ideas defined by building the Grade application is presented. Some further implementation specific ideas are presented in a following section.

To identify the desired result of this section see Figure 4.1. On the left side a typical QVT-R code is placed that is visualized in TROPIC on the right side. This example is used in hereinafter sections to explain how this functionality can be achieved and why it needs to be displayed as it is done in this figure. Several more specific cases are added by additional examples.

The desired integration of Grade in the model transformation process is visualized in Figure 4.2. First, syntactically correct QVT-R code has to be written and a certain input `model` needs to be defined. These pieces of information are forwarded to Grade (the grey box in the background of Figure 4.2) and transformed to a TROPIC Net in order to visualize the operational semantics. This Net can be analyzed in the **Debugging View** by stepwise proceeding the transformation. By identifying an error, changes are undertaken at the QVT-R code. On the left side the **QVT engine** is placed that consumes the QVT-R Code – holding the knowledge about the involved source model, source metamodel, and the target metamodel – and produces the target model conforming to the target metamodel. The **QVT engine** itself is not part of Grade.

4. Transforming QVT Relations to TROPIC

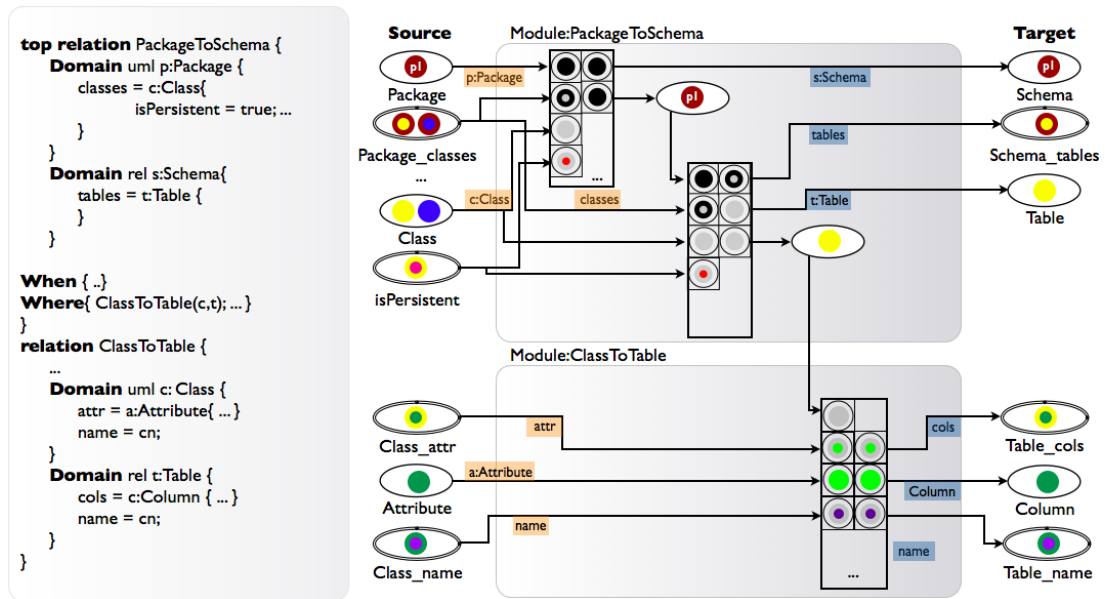


Figure 4.1.: A basic TROPIC visualization of a typical QVT-R snippet

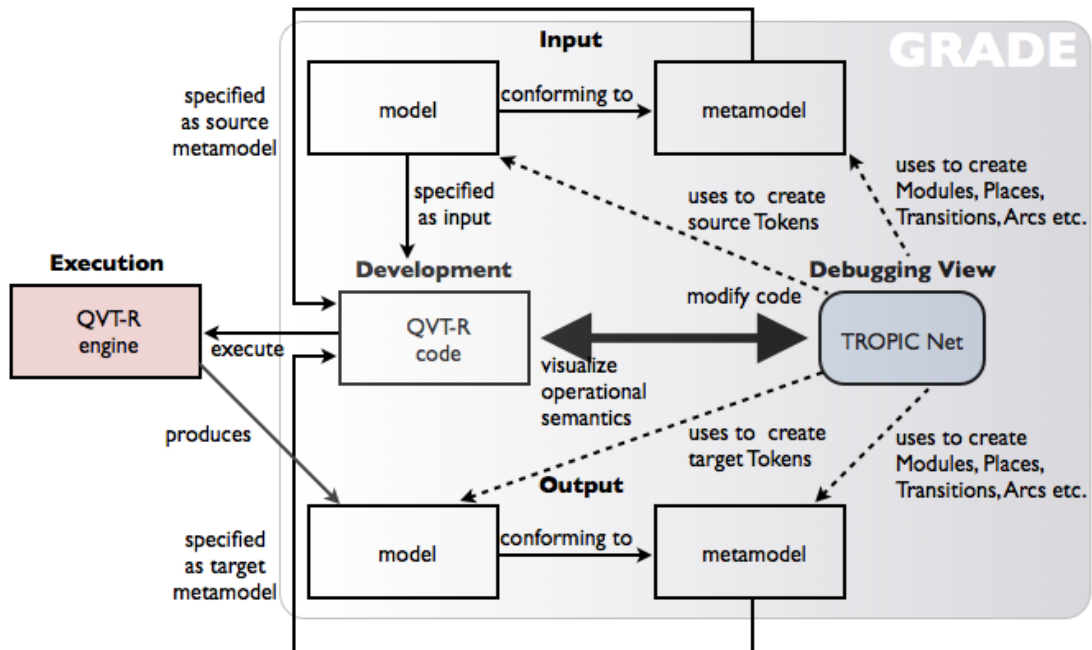


Figure 4.2.: The interaction of Grade with other components and artifacts

In particular the process necessary to receive the Transformation Net from a QVT-R file and input model is treated in this conceptual approach. How such resulting TROPIC nets can be analyzed is introduced in the section of evaluating the results.

4.1.1. Execution Semantics of QVT-R

In this section it is the aim to specify in which ordering Relations can be transformed to TROPIC by building a graph that specifies paths that can be followed. To create such a graph, an entry point has to be found. Obviously, a tree like situation with a single root-node and always two nodes (as balanced tree) that are directly derived from every root would be perfectly for a transformation process, but unfortunately the situation is more complex.

QVT-R introduces two different types of Relations, as already mentioned before. These types reflect the hierarchy level of their execution in the process of visualizing them. Top Relations mark possible starting points of the graph, whereas non top Relations can only be called by a top Relation in its where or when clause (or by a Where-Clause in a Relation that has been called before by a top Relation). Therefore, it would be possible to build multiple trees that coexist in parallel, but do not interact. Again it has to be more sophisticated. Although top-Relations can be directly called, preconditions can avoid their execution (When-Clauses). Therefore, it can only represent a root tag for a following computation path. After in all paths all nodes have been computed, such a Relation could be useable. In the common sense of terminology Relations with preconditions are illegible for being used as root nodes.

Another case is the problem when one top Relation calls a Relation in a Where-Clause that has a When-Clause expecting another Relation of another tree to be already executed. At this point merging the two trees does not make sense anymore and probably does not work. Hence, a graph results with multiple root nodes being able to join at some points. This is simplified in Figure. 4.3.

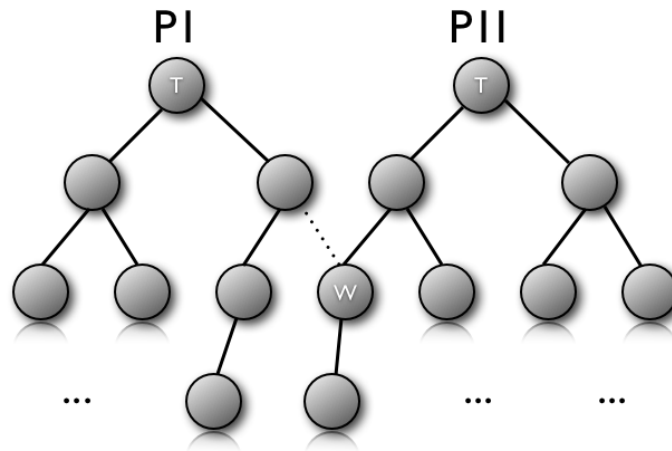


Figure 4.3.: A graph representation of QVT-R Relation ordering and its execution paths

In Figure 4.3 there are two main trees PI and PII. Both sides start with a

4. Transforming QVT Relations to TROPIC

top Relation marked with T. All edges linking derived Relations can be seen as Where-Clauses in the QVT standard. Although the main flow directs from the top of a tree to the bottom, undirected edges are used, because it is very valuable to be able to go back to a previous node. The node marked with W represents a Relation that holds two undirected edges from parent nodes. One of them is visualized with a dotted line. This represents a When-Clause and is, therefore, no direct dependency. This means that the parent node reachable over this path does not call this child. To be able to execute the child Relation, however, the parent item execution has to be completed before this transformation can be started or it has to be postponed for later execution.

This example shows how the classification of Relations could be seen in an abstract way. There exist two trees that are not connected to each other by a direct path. Only a Where-Clause forms a dependency that has to be fulfilled. As this example shows it could be assumed that the left tree could be integrated as a subtree under the node of a Relation with a When-Clause. However, if another When-Clause at another point exists, that points to another Relation – even to a parent node of the Relation with the When-Clause or to another tree – this cannot be done anymore. Another problem could be seen, if a When-Clause could never be fulfilled, because the referred Relations is never called. This is a severe computation problem which does not exist for When-Clauses as they are always directly called in the parent Relation. Furthermore, it can be seen that the Where-Clauses form the edges of the graph and, therefore, are – beyond the entry points – the items defining the execution path. This is helpful, because Where-Clause directly pass values to following Relations in QVT-R. This can be used in the computation as well as for both Relations the information can be made available. This need not hold for other approaches.

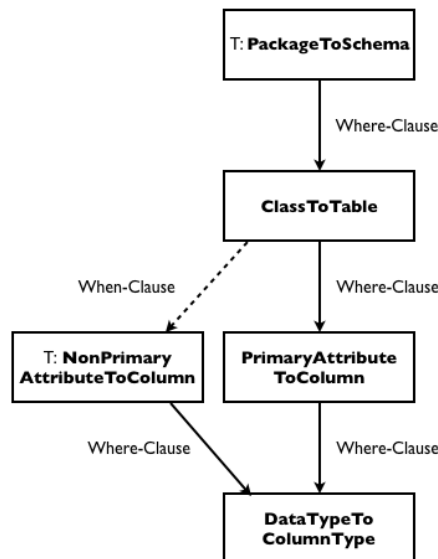


Figure 4.4.: An example graph using two top Relations that involve several Where-Clauses and one When-Clause

See Figure 4.4 for a concrete example of using several top Relations with Where- and When-Clauses. The Relation `PackageToSchema` and `NonPrimaryAttributeTo`

are top-level Relation marked with the letter T. These two Relations can be considered as entry points for the execution. However, at a closer look a precondition using a When-Clause at the `NonPrimaryAttributeTo` can be identified. This Clause requires that the `ClassToTable` Relation has been executed before. This cannot be the actual state by starting the execution. The only way to fulfill this requirement for a top-level Domain is to start with another Domain – in this example this can only be `PackageToSchema`. If there does not exist any further top Relation, the execution cannot be continued. After starting with `PackageToSchema`, `ClassToTable`, then `AttributeToColumn` and finally `DataTypeToColumnType` are executed by calling these Relations using a Where-Clause. After this tree has been executed, the next root node `NonPrimaryAttributeTo` is tried to be executed again. At this time the precondition is fulfilled and the execution can continue. Furthermore, this Relation involves the second calling of `DataTypeToColumnType` which need not be redundancy, because it can involve a different set of objects (here non-primary or primary).

So first the Relations are categorized in top-Relations, top-Relations with When-Clauses and all other Relations. The transformation process clearly starts with the top-Relations without any precondition.

By taking one Relation the need can arise to execute related Relations (Where-Clauses). It could make sense to assume that all nodes at one hierarchical level are somehow equivalent to each other. Therefore, it sounds logical – although this is fully in the responsibility of each specific model – to assume that by executing a Relation the probability of a precondition of another Relation at the same level is fulfilled is higher than by executing just “another” Relation. This leads to an approach similar to Breadth-first-search through this graph. However, this idea is more problematic than the deepening over Where-Clauses. Following Where-Clauses has the advantage that the developer of the QVT-R Relations is of the opinion that the called Relation can directly be followed by the calling Relation which improves the probability of being able to execute them without involving intermediate Relation executions.

For example in the Listing 4.1 the Relation `ClassToTable` holds a Where-Clause calling a referred Relation `AttributeToColumn` with the variables `c` and `t` as passed attributes. Furthermore, it is possible to call several other Relations or the same Relation with different data stored in different variables.

Listing 4.1: QVT Relation with a Where-Clause calling a referred Relation

```

22
23 relation ClassToTable{
24     cn: String;
25     checkonly Domain class c:Class{
26         name=cn
27     }
28     enforce Domain rel t:Table{
29         name=cn
30     }
31     where{
32         AttributeToColumn(c, t);
33     }

```

34 }

To be able to get the data that is passed within a single parsing step, it is necessary to execute a Relation included in a Where-Clause of the current Relation directly after the execution of the current Relation itself (Depth-first-search algorithm). Furthermore, this concept also helps to improve the performance, because at the point a Relation is called as post-condition the Relation should be ready to be executed. The big disadvantage of the used algorithm is clearly the insolubility of infinite Relation-dependencies. Nevertheless, this problem exists for other algorithms as well, because in EMF the serialization of models – which is necessary to be able to store them – is accomplished in a single writing process. Therefore, a resulting Net which is produced using EMF like in this thesis, cannot provide any partial results and, therefore, has to fail for graphs of infinite lengths. So following this path – a path to the depth by using referred Relations in post-conditions – provide higher probability of executability with high performance. The actual call and execution of the Relations are done by recursively processing from the top of the virtual graph down by calling the Where-Clauses. The Where-Clauses are executed like normal Transitions, but holding some external data – the parent Relation. After the last Relation of a path has been reached the next path down from this top-Relation (“root”-node) is executed, if existing. Although Where- and When-Clauses can often be substituted in some way (a Where-Clause of level N can be seen as When-Clause at the level N+1) there can be the case both coexist in just one Relation. For example a top-Relation that holds a When-Clause can be necessary to force the execution of this Relation, but it, furthermore, implies the usage of a particular ordering. The same top-Relation could also be called from another context by a Where-Clause, meaning that in the case that the Relation is reached there arises the need to execute another Relation. A top-Relation, however, always has to be called at least once. For such Relations with unfulfilled preconditions that were called as post-condition of another Relation lead without fail to a situation in which an execution cannot be continued. For this case the Relation is stored on a stack for later execution.

After all top-Relations were tried to be executed, all top-Relations with When-Clauses are processed. The process works exactly as described before. If some Relations are not able to be executed, although they have been attempted to be executed, they can be found on the stack for later execution. This stack is the last to be processed and continues until all nodes were successfully transformed.

For the case a precondition is needed that does not exist or cannot be fulfilled, an error is produced. For such cases tools like “medini QVT” that can execute QVT-R code provide error messages. Therefore, this thesis does not focus on syntactical correctness, but rather on correctly producing what the developer expects by formulating a mapping rule.

4.2. Types

It has to be distinguished between primitive and complex type objects – such objects can represent certain root elements that need to be called or properties that are place in root objects. Primitive type objects hold a certain value or

variable, whereas complex types hold a pattern similar to Domains (Domains hold a Domain pattern that uses the same complex type). To execute a primitive object a direct creation of Places and Arcs can be done. However, the complex types recursively call the same method in the code as for the Domain pattern execution. This is necessary because such complex objects can be deeply nested (complex objects holding complex objects) – even infinite paths or loops can be constructed – and should be executed in the same manner as they have the same issues that need to be handled. The execution of a Domain and its contained properties only stops at the point in which no primitive or complex property – including its child elements – in the Domain is unexecuted.

Moreover, there is the need to differentiate between objects used as conditions (one-sided) and two-sided constellations passing values from the source to its target representation. One-sided variants represent conditions like `isPersistent=true` that request a certain condition for a certain object mapped by a Domain to be executed with this Domain. Another alternative is that it just creates Tokens for the target model that have this certain property. This implies that there is no direct relationship between a one-sided condition in source Domains and other ones in target Domains – semantically, however, this could be intended to hold. In contrast two-sided variants provide certain linkings between input and output property values. For example `name=cn` (see Figure 4.1) with a variable named `cn` used in both source and target Domain results in output Tokens that have the same name as its representation in the source Tokens. The need for such two-sided definitions is based in the opportunity of using bidirectional usage in QVT-R.

To be able to map the different types of objects, it has to be defined why to use two-coloured or one-coloured Places. Two-coloured Places represent a certain dependency on another (typically in hierarchically superior position) element – such as **Package** to **Class** in Figure 4.1. A complex type is dependent on another parent object or the Domain holding such properties. In the QVT-R code it is easy to differentiate between these types. Braces with or without same mappings within them are always an indicator for a complex type – see `t:Table` in Figure 4.1. Whereas a primitive type holds some simple value like a certain character string, variable, boolean or other value – like `isPersistent = true`.

Each type – complex and primitive – belongs to one execution side – the source side that has mainly reading functionality whereas the target model side that produces elements. So, it is essential to differentiate between these two different situations. For both the so called EndpointModule – the source or the target Module – has to be defined or looked up. These Modules are stored in the basic data set of the execution process and are called **Source** and **Target**. More about the storing, definition and execution process is described in the following section about the practical realization. Moreover, it seems notable that the usage of MetaTokens in Transitions - and they can only be used in Transitions - involve the usage of In- and OutPlacements. These Placements do not provide more functionality as providing a base for placing MetaTokens in a Transition. In the context of this master thesis never more than one MetaToken is placed in a Placement and it is never intentionally done to add Placements without MetaTokens. This is typically done in one encapsulated process. Hence, for every

4. Transforming QVT Relations to TROPIC

MetaToken a Placement according the execution direction has to be placed in the certain Transition and is left out for easier terminology and better overview of the described concepts.

4.2.1. Domains

Domains are the most essential element within a Relation that specify concrete constraints that need to hold or objects that are created. Domains can be checkable or enforceable as described in a section before. To transform a Domain it is necessary to understand the basic concepts. A Domain holds a Domain pattern that posses the data of the Domain (the differentiation of the Domain and its pattern is clarified later on). Furthermore, a Domain pattern holds some properties. The Domain itself provides the necessary execution information. In Figure 4.1 the Domains of the Relation `PackageToSchema` are transformed in the Module of `Module:PackageToSchema`. The actual mappings like from `Class` to `Table` are declared in the pattern of the Domain. At this point it seems notable that in the context of this thesis the usage of `Domain` typically refers to the Domain and all included objects like Domain patterns.

The processing of Domains has to start with the determination of execution direction. This decision has to be split into two possible situations. On the one hand unidirectional QVT-R Relations may be used. Such Relations typically use the keywords `checkonly` and `enforce` – e.g., in Figure 4.1 the Domain `p:Package` is `checkonly` and `s:Schema` is `enforceable`. For this reason the execution direction obviously has to be established from the Domain marked as `checkable` to the Domain marked as `enforceable`. Two Domains that are declared to be checkable and enforceable are not considered in this thesis, because this represents a syntactical error that should be recognized by IDEs. On the other hand both Domains could be enforceable. Therefore, it has to be read from the Domain to which model a Domain belongs to. Nevertheless, it is not enough to know from which or to which model a Domain belongs to, it also has to be defined which of both models is the “source” and which is the “target” model. Execution engines integrated in an IDE like medini QVT also need this information to be able to execute a MT. Hence, it is necessary to get this information from the user. In this thesis only the second variant is used, because it is broader usable and cleaner in its definition.

The particular transformations in the Domain are done in the Domain pattern which is a complex type as well. However, the Domain itself is at some points different to other complex type objects usages. First, it represents a root tag within a Relation, whereas other complex type objects are added as properties to the Domain pattern (hierarchically deepening) and, therefore, only depend on their parent objects. The Domain pattern and all contained properties depend to one concrete execution side (source or target). So this circumstance needs to be identified and stored on the level of the Domain pattern to be used later on. This execution side influences the placement of the Places – they have to be placed in one of the EndpointModules. Furthermore, the direction of the Arcs and the usage of In- or OutPlacements in the Transition have to be adapted to this fact. The accomplishment of the concrete mappings for the Domain pattern

are described in the section about complex types. The colouring necessary to visualize the operational semantics in TROPIC is expressed in the section about correspondence identification methodologies.

In addition, if the Relation holds a When-Clause that specifies the variable of this Domain or one of its properties has already been created, then it should be desisted to create a one-coloured Place for this item. However, a two-coloured Place combining the variable of the parent Relation with the instance in the child Relation is needed to express the coherence.

4.2.2. Primitive Domains

There is often used a so called primitive Domain [19] which was not specified in the QVT Relations Standard 1.0 [22]. Primitive Domains are Domains that possess a data type, but no specification of the Domain pattern. Primitive Domains only have the purpose to forward the values of QVT data types at the call of Relations without any proof of provided conditions [19].

In contrast, the evaluation of the contained mappings of classical Domains have to be proofed and translated. The passing of values for both variants can be accomplished by using Where-Clauses.

See Listing 4.2 for an example using primitive Domains. The primitive Domain is called `prefix` and is of the type “String”. Other variables or constants can be stored or added to this primitive Domain and can be reused in following Relations.

Listing 4.2: A QVT Relations example with primitive Domains

```

36 top relation ClassToTable {
37     cn: String;
38     domain uml c:Class {
39         name=cn
40     }
41     domain rdbms t:Table {
42         name=cn
43     }
44     primitive domain prefix:String;
45     where {
46         prefix = if (cn = '') then '' else cn endif;
47         AttributeToColumn(c, t, prefix);
48     }
49 }
50
51 relation AttributeToColumn {
52     pn: String;
53     domain uml c:Class {...}
54     domain rdbms t:Table {...}
55     primitive domain prefix:String;
56     where {
57         prefix = if (prefix = '') then pn else prefix+'_'+pn
58             endif;
59     }
60 }

```

```
59     }
60 }
```

The primitive Domains are not used in the further proceeding of this thesis and, therefore, the name Domain always refers to the complex variant.

4.2.3. Primitive types

Primitive type properties can be seen as simple types holding a particular value like the boolean `true` of `isPersistent=true`. Primitive types have to be placed within complex types such as Domain patterns.

Primitive type properties can be translated to two-coloured Places on Source and/or Target side. The two-coloured Tokens that are added to these Places hold the colour of their parent type – the parent type must be a complex type like a Domain - and its own colour value. Therefore, this represents the linking between these two sets of data and assures that dependencies are expressed. For complex type this transformation has to be more complex unless more coloured Tokens are introduced.

The primitive types in the example of Figure 4.1 are `name` and `isPersistent`. `isPersistent=true` holds a certain value that does not correspond to any value on the other side (target or source). All other classes are ignored for this transformation rule. Such conditions could also be used to set a standard value that is needed for all transformed Tokens independently from its value. Therefore, such conditions are only represented in Transitions by one InPlacement or OutPlacement respectively. The property `name` is a variable expression. The variable `cn` creates the possibility to link the values of both transformation sides. This notation is used to be able to transform QVT rules in both directions without changing the code. Hence, the usage of such expressions implies the necessity of an In- and OutPlacement for each of them. So this can be seen as the classical **simple** transformation where data from one model is passed step by step to the target model.

The relationships of primitive and complex type properties can be seen in the graphics of Figure 4.5. As already mentioned primitive type properties can be strings, booleans or other types. They always have to depend to a surrounding complex type property which itself can hold several different other properties.

4.2.4. Complex types

Supplementary to primitive type properties there are complex types like the Domain pattern of `Class` of Figure 4.1 or other nested complex types. Complex types can be seen as a similar concept to objects or their classes in object-oriented programming. For example in the Domain `Class` another class could be instantiated – independently from its meaningfulness – without any specified data – `cl:Class`. This means that an isolated Token – a Token unrelated to any other Token such as representations for attributes on the same execution side – is created for the target model. In many situations these created objects are passed to other rules and refined there, but also empty objects could make sense. Therefore, for the transformation it has to be considered that another set

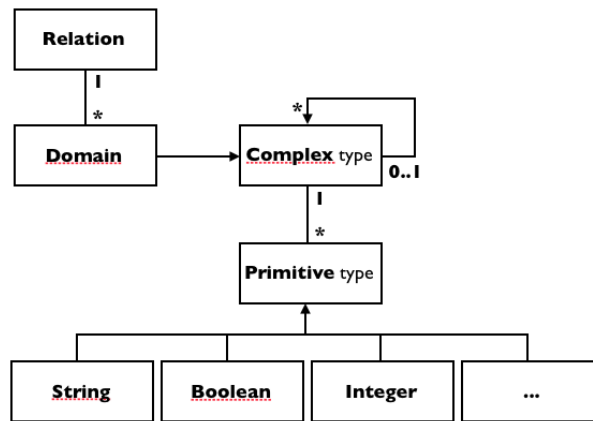


Figure 4.5.: A model of the relationship between simple and complex types

of data is used in the complex types. Additionally, it seems notable that Domains itself are no complex types itself, but they hold a Domain pattern that actually is a complex type that represents all contained nested mappings. Therefore, Domains with their patterns are specialized complex types, as they represent entry points within a certain Relation (this does not imply that information cannot be consumed from other Relations). All other complex or primitive types can only be included and encapsulated in Domains (the owners of the Domain patterns). So, the Domain pattern is the container for the mappings. The Domains are responsible for storing general information like checkable or enforceable and, therefore, are the holders of the meta data that is necessary to execute the contained pattern and its properties. It seems notable as well that in most cases it is not differentiated technically between **normal** complex types and a Domain pattern, but the TROPIC language does need this differentiation for a correct transformation. See Figure 4.1 to get an idea about the different interpretations – the example is explained in all details in a following section.

Hence, this implies that there can never result an empty Transition from any Domain definition – even empty Domains (empty brace block). Consequently there cannot be an empty Transformation Net (or Module of a net), if at least one Domain is used – according to the typical usage two Domains are necessary. The QVT Parser mentioned above does not differentiate between Domain patterns and other complex types at all, but in this master thesis this needs to be done for addressing entry point constellation which intensively affects their execution process – this can be seen in the discussed example of Figure 4.1.

The mapping and colouring of Places is more complex than in primitive type variants. To map a complex type like a Domain to TROPIC a one-coloured Place is needed to hold its value. But for every complex type that is placed within another complex type, a two-coloured Place for the representation of dependency and a one colored place for the referred complex type is needed. Both Places are added to the EndPointModule. This is similar for all other nested complex types, because their properties clearly and visually have to belong to them. So, to express such a constellation at this point some own data of this type is needed to be passed to its child. For sure a two-coloured Token can hold an own value as well, but at this point it has to be stated that it is unnatural that similar

4. Transforming QVT Relations to TROPIC

types (Domains and enclosed complex types) are translated differently. In the context of graphical debugging it is most necessary to overview the classification of different types and their handling. Therefore, it is important to be able to track the proceeding of such a complex type. One-coloured Places allow the focussing on the most important issues. It always needs to be clear which properties belong to which complex type (including Domains). This is done by correct colouring in this thesis and leads to the usage of two-coloured elements for such situations. Even for constellations ranging over several Transitions these dependencies using their colouring have to be kept.

For example, if the complex type `Class` is transformed and passed from its origin using postconditions – Where-Clauses – from one Relation to another to refine it. In such transformations there is probably no need to use the data of the parent type unless it is not passed itself. Hence, a visual differentiation makes sense. A parent-child synthesis could be a two-coloured Place called `Package_classes` combining a certain Package with a Class.

For all created Places so far an InPlacement for types of the source model or an OutPlacement for other types respectively – including correct MetaTokens – have to be created in the current Transition. Afterwards Arcs can be created from a source Place to an InPlacement and from the correct OutPlacement to the receiving Place. The details of when which MetaToken has to be used is specified later on.

As a consequence complex types with the enclosed other types and the needed transformations tend to be enormous complex – not only in visualization, but also for QVT engines to execute them. So, from performance side it is, obviously, important to minimize complex type relationships (especially in one-to-many constellation as discussed later on) in favour to primitive type variants. The Grade debugging mechanism presented in this thesis helps to discover such performance leaks. Of course, in many cases a change from complex types to primitive types or a change from one-to-many to one-to-one is impossible, but from debugging side it is only the issue to highlight possible problems. The clue has to be made by developer himself or herself.

For better understanding the nesting of complex types within another complex type (e.g., the Domain pattern) can be seen in Figure 4.5. This figure clearly systematically highlights the broadening and deepening effects of nesting complex types.

4.3. Relationships

The relationships used in MOF models are two-sided which means that multiplicity can be defined for both reading directions. In this thesis a TROPIC Net for QVT-R debugging always represents a hierarchical dependency. This means that only a single parent object for a specific child object can exist, but several child objects can exist per parent object.

Relationship variations It has to be differentiated between one-to-one- (1:1) and one-to-many-relationships (1:n). The former ones are easier to handle, because it is obviously true that a certain object of one complex or primitive type

directly depends on the instance of the enclosed complex type. Therefore, there is no need for a new Transition and a TracePlace as an intermediary space to wait for the further execution. However, 1:n-relationships – see **Package** and **Class** being transformed in the example of Figure 4.1 – have the problem that a number of other types can depend on one complex type instance. So, a new Transition is needed that is dedicated to match these one-to-many-relationship transformation. A TracePlace combines the outgoing instance of the parent type with the new Transition. The reason for TracePlaces in this context is the fact that the colour of one parent complex type is needed in all of its children. Because 1:n-relationships imply that more than one child can exist for this type, the parent type must be made reusable. Moreover, another execution part is fired several times defining all the subelements. A duplicate from the OneColorMetaToken representing the **Class** is placed in the new Transition to receive the information from the TracePlace. Another two-coloured Token is needed here for the synthesis of the passed Tokens from the parent type (TracePlace) and the nested 1:n-related properties. These two-coloured Places are addressed in both Transitions, because the parent Transition should only be fireable, if at least one child object is available. The child Transition has to use address these Places (by using MetaTokens) to verify that for a linking between the passed value from the parent Transition and a child object exists.

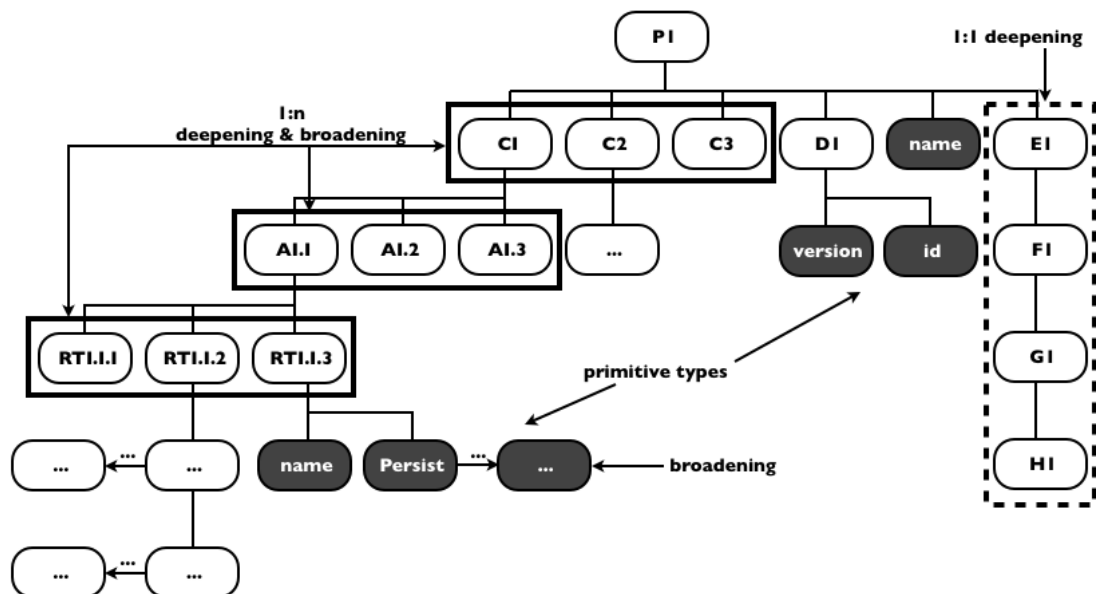


Figure 4.6.: A combination of one-to-many and one-to-one related properties

As exemplarily shown in Figure 4.6 one-to-one and one-to-many related properties can both be used within one complex type. However, 1:n related items can involve several different referred instances and, therefore, are displayed as range of instances (box with black border). Every instance of a complex one-to-many related property can hold again other 1:n related properties according to the used metamodel. Therefore, this visualization broadens and broadens. Meanwhile it can deepen and deepen as well, but in Figure 4.6 the tree downwards node **E1** shows that the missing broadening leads to easier manageable situations.

4. Transforming QVT Relations to TROPIC

An example of a TROPIC representation with correct colouring can be seen in figure 4.8 which is explained in more detail later on.

TracePlaces for 1:n relationships As described before TracePlaces are used to allow the usage of 1:n-relationships. Nevertheless, it is necessary to acknowledge that these TracePlace-situations do not place the incoming information as main information / main type of the following Transitions. It is rather used to create a linking – two-coloured Tokens, MetaTokens and Places - between the parent and every child instance. The type of the property that forces the execution in another Transition is set as main type for this Transition instead of using the parent information like for Where-/When-Clause-TracePlaces. This is very essential for the handling in the implementation.

4.4. Dependencies

Besides the introduction of classical relationships on metamodel level and, therefore, on model instance level as well, the dependencies resulting from QVT-R rules definitions need to be translated. These dependencies allow the breaking up of long transformation rules (Relations) into several depending sub-rules or the transformation to TROPIC elements for dependencies resulting from the involved paradigm change.

4.4.1. Where- and When-Clauses

The Where-Clause combines two different Relations with each other. Hence, it has to have a MetaToken in the calling Relation's Transition that receives the data from a TracePlace. The target side does not receive information, but produces it. The TracePlace does not receive the data from the Place in the EndpointModule, but from the parent Transition. Therefore, the OutPlacement that holds the main type of this Relation is duplicated with its contained OneColorMetaToken combined with a TPArc with the created TracePlace. So far is not clear which information of the previous Transition is passed to the child Relation. This needs to be identified which is explained later on. In contrast TracePlaces resulting from 1:n relationships automatically provide the parent type and the Transition in which is transformed.

The main type is typically the type of the Domain pattern, but could also be a one-to-many related property that involves a new Transition. Therefore, it is essential to take the last hierarchy level. Combinations from every level of each Domain are not necessary. The concept of different hierarchy levels is out of the scope of this section and is explained later on. Going back to the original problem of combining the two Relations, the duplicated OutPlacement has to be combined with a TPArc with the created TracePlace. As a last step the TracePlace has to be registered for other execution paths that might want to reuse it.

The When-Clause execution is similar but not the same as Where-Clauses. The origin has to be looked up. The Relation calling another Relation in a Where-Clause can be directly linked. The When-Clause is a little diverse for this issue. It can refer to direct execution “neighbours”, to another Relation with longest

path in-between both Relations, or even to Relations that have not been executed yet. Hence, this can only be executed in the called Relation (there is not even a calling Relation). Implementing this issue the differences is bigger, because the Where-Clause can be handled in the call of the referred Relation. At this point both Relations are presented and their information is known. For When-Clauses the execution in the code can not directly match them together and so it is necessary to find out which Module with which Transitions belongs to the certain Relation. Therefore, these pieces of information have to be stored and managed.

In both constellations the information passed over the TracePlace to a child Relation can be considered as the main information for this Transition - its type, therefore, can be seen as main type that somehow “owns” the Transition. This is important for the colouring later on and reflects the differences in translating these TracePlace-situations to all others – in respect to the linking mechanism of the two related Transitions.

As already discussed 1:n relationships always force at least one child object is produceable (hierarchically deepening). For Where-/When-Clause this is not necessary. This weaker specification is adequate, because the data is passed on to the another Relation and does not expect anything else then the usage of this data. There is no dependency in the other direction that could force the parent type not to execute. So it has to be assumed that the execution of the previous Relation has been successful according to its intended and defined transformation mappings.

4.4.2. Hierarchical Data Passing

A Where condition means that the passed values have to be executed in a called Relation. For this reason the value that it is passed on has to exist or has to be created in the calling Relation. The value of the passed variable is needed to continue execution in this particular instance of data. It is the normal flow that a Token is passed from a source Place to a target Place - or just represents a certain condition for the incoming or outgoing Token. However, in this case the data passed in Tokens is not only needed at the target side, but also in the referenced Relation. For example to transform a Class with Attributes it is necessary to pass the classes to the Relation that executes the attributes for this class to be able to create a two-coloured Place and a two-coloured MetaToken in the Transition. The two-coloured items are needed to visualize the linking between parent and child constellations. Such a constellation can be a Domain to a property or a complex property of the pattern of a Domain to its properties or just a passed variable in a Where-Clause.

Additionally, there can exist the situation of variables being hierarchically passed over several post- or preconditions. For example the Relation **ClassToTable** could create an attribute. This attribute is passed to a Relation determining which type of attribute it is and passes it on to another Relation that fills this attribute with data. Then the creation of TracePlaces would result in several Places that do nothing then passing a value. For the first Relation the attribute is only a precondition. The second Relation really needs the data to

4. Transforming QVT Relations to TROPIC

create the mappings. Hence, a great number of TracePlaces could be created, but in reality there arises the question, if it is sufficient to link the original TracePlace with all others waiting for the data within one execution flow of Relations. The definition of this theory also implies that is obligatory to search for already existing Places before creating them. To be able to handle this situation with better performance, in this approach every Relation registers its used, created and known TracePlaces. This means that the same TracePlace could be stored several times like in the example mentioned above (Listing 4.1). Thus, it is possible to retrieve passed data by knowing its own parent Relation, because this parent Relation does hold all hierarchically passed values. To be able to supply following Relations again with this full set of data, it makes sense to automatize this registration procedure. However, this theory would lead to a situation of hierarchically passing values from one Relation to another and again to another or even more without involving all defined conditions and issues in the original QVT Code. Just think about the situation that one Relation creates a **Class** and passes it to a Relation that adds some properties to this item. Furthermore, the called Relation calls a third Relation and passes this variable of a **Class** again on. By passing the data hierarchically and directly from a source Place that lies in between the first and the second Relation only consuming the data of the first, this would lead to a situation in which the third Relation could be fired more often than the second Relation executes. Although the result of Relation two (at least in the context of **Class**) should be a dependency for further executions. Consequently a hierarchical passing cannot be implemented to fulfill all restrictions. Therefore, a similar dependency on each level of the computation graph of QVT-R is to interpret as (potentially) different instance to following levels, although they might involve the same set of variables.

Nevertheless, there is a possibility of sharing TracePlaces among different calls of referred Relations. It is often the situation that one Relation calls several other Relations. Such constellations should not result in several TracePlaces, but lead to a sharing of TracePlaes as they are involving value passings on the same hierarchy level.

In the context of Grade the creation of TracePlaces is placed on Relations level. This is done, because at this level the Where-Clause information is easily extractable and analyzable. However, the usage of these TracePlaces is rather done in the transformation of Domains, because Domains hold the data that is transformed and offer the possibility of lazy combining the two Relations. Another tested approach is the usage of two different stages of TracePlace transformations. On the production side there is the possibility of creating a TracePlace and the connection from the calling reference at the time of recognizing related Relations. On consuming side the TracePlace is looked up (in a list stored in the application) and combined with the actual Relation. This would lead to the possibility of looking for errors in transformation by searching for unconnected TracePlaces. A reason for this could be a called Relation via Where-Clauses that has some When-Clauses (Preconditions) avoiding its execution until the end of transformation. This can only happen, if another Relation specified in the When-Clause can never be fired, because it has not been defined top-level and has not been called before (Where-Clause). However, this approach could not be used in

the practical solution for two reasons. On the one hand the data handling is much easier, if both sides are known. The colouring can be done at once, combining both meaningful. Other MetaTokens need the colour of the incoming MetaTokens. On the other hand the ease of debugging of the source code increases by combining semantically related steps. Hence, the unusual case described above is ignored. Nevertheless, this fact can be changed easily in the future.

4.5. Finding Correspondence of Source and Target Objects

One of the most important points is to define which item of the source model depends to which item of the target model. This means the identification of correspondences (or equivalent structures). For humans it is, obviously, semantically true that a class is related to a table, but syntactically – and all computers rely nearly only on syntax and definitions – this relationship cannot hold without any further information.

4.5.1. Heavyweight Approach

For reason of correspondence identification it must be analyzed which items belong together by stepwise parsing the Relation. First of all the Where- and When-Clauses are relevant, because they offer an easy overview of variables that can be linked. Just look at the Where-Clause in the example of Figure 4.1.

The Variable *c* is matched with *t* and passed on to the following Relation. Therefore, for this and for the following Relation these two variables have to be equal. This cannot be guaranteed for any other variable involved. The Where-Clauses are syntactically the same. A combination of two variables in a When-statement of a Relation must hold for this Relation and for its parent Relation. Parent Relation in this context means the Relation specified in the When-Clause.

Another construct used in QVT-R is a primitive type with a Variable. This variable implies a two-sided usage and in this context affords to define that the properties calling this variables are equal, but also all other properties on hierarchically higher levels. This means that two complex types such as Domain patterns are parsed. With every iteration both sides go deeper in parallel and look for variables. All passed complex types are stored to be able to track which items were passed. Finding another complex type means that the iteration has to go deeper. If a variable is found, the variable is stored with its stack of passed properties. The same is done for the other side. Afterwards the stack of properties is read backwards and matched one-by-one. With this backwards-approach it is unproblematic, if the list is longer on one side than on the other side, because entries without counterpart are just left out and have to be matched somehow else.

The third step in the computation of Relations is to check, if only one complex type is available on each side. If this is true, these items obviously belong (or can belong) to each other. Therefore, we can map them as equal. If more than one type is available per side, then it has to be considered, if previously known

4. Transforming QVT Relations to TROPIC

correspondences (equivalent structures) can help to solve this issue. Otherwise a many-to-many computation of all paths would be necessary leading to wild permutations and heavy computation problems. This is the only reason why Where-/When-Clause analyzation and the examination of variables have to be undertaken in previous. Furthermore, this step can even provide more advantages. Instead of checking only available types, it is possible to check the multiplicity of arguments. If only a single one-to-many or one one-to-one is available on each side, then a correspondence seems to exist. Hence, this “weaker” condition can lead to powerful results showing further equalities.

All these discussed analyzation steps have to be stored as useable objects. There always exists the possibility that some names occur in different properties with different meaning. Therefore, the name is only interesting within one property and cannot be used to check equalities of types. In the practical part of this thesis, a special data construct was created called GradeTable. A GradeTable is a double-sided hashed two column table that stores a complex type as key and all other known equal types in a Set (Sets do not hold duplicates!). This allows the fast querying of values of both columns. So for realizing an equality between two variables, lists of equal types for both instances are retrieved without any further handling. The avoidance of duplicate values is important as QVT-R handles variables name like other languages as unique specifiers for variables. Furthermore, it is necessary to remember all Transitions and MetaTokens involved in transformations with this variable. Therefore, tables for these issues have exist as well.

Another problem has not been discussed yet. If more than two Domains - one for each side - exist, all pairwise permutations of these Domains have to be checked in the previously discussed manner. This would involve an exponential computation effort increase. The minimum Domain count per Relation should not be less than two to be able to transform source and target side issues. Additional Domains do not specify information indescribable in two Domains, they only split the execution side in several Domains per Relation. Hence, we can assume that only two Domains are used to reduce the effort dramatically (Assumption I). In the OMG standard no example is mentioned that violates Assumption I and for real world problems typically it holds as well.

In Figure 4.7 the basic structural discovery of correspondences is summarized by identifying structural equivalencies. On the source (left) and target (right) side three complex types exist being linked by deepening 1:n relationships. The QVT code is placed in the center of the figure, because it is responsible for the transformation from one to the other side. Correspondences can now be discovered by counting complex types in each parent complex type and comparing it with the correspondent target element. However, it might also be necessary to deepen the counting of complex types (beginning at the root element). This can lead to an inference of relationships. Another discussed strategy is the search of variables (cn in Figure 4.7). If one variable occurs on each side, this represents a structural equivalence between these two primitive types. Moreover, it can also mean a correspondence of the path leading to this element on each side. Similar is the analysis of Where- and When-Clauses that express equivalences by mapping two types/variables together. The inspection of the path to reach the elements on

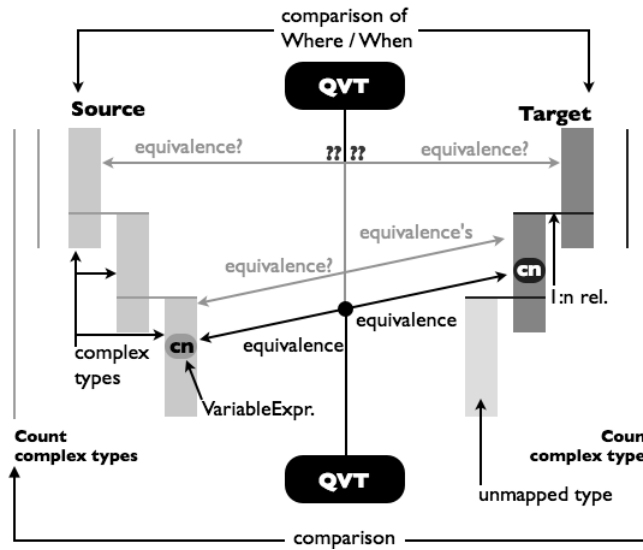


Figure 4.7.: A structural summary of discussed equivalence discovery

each side are as interesting as for variables. Another interesting issues of Figure 4.7 is the **unmapped type** on the target side. This type seems not to correspond with the source side, because the correspondence inference deduced from the analysis of variable **cn** made it very probable that the parent path of each side involve equivalent structures. It is from this perspective not provable which role the **unmapped type** takes in the model transformation. Further pieces of information have to be gathered to be able to construct an equivalence relationship. Naturally, this type could also be a newly created type that does not correspond in any way with the source side besides the embedding in its parent type that holds a certain relationship.

4.5.2. Lightweight Approach

Using hashed data structures and using Assumption I can highly reduce the amount of computation necessary and can help improving the quality of execution. However, the effort producing linkings in the discussed manner is still high and there are still open questions preventing an execution for 100 percent of all QVT-R transformations.

The gaps result from new Transitions that are created for each 1:n relationship or even for one-to-one constellations. To be able to link them, data has to be known, but to be able to execute the second Domain representing the second side (target side) of the Transition, it has to be known to which Transition it belongs to – if there exist several Transitions produced by the source side, the target execution side does not know to which Transition of this Relation this execution item semantically belongs to. If there exists only one Transition, this is obviously easy. So one-to-one related properties are less problematic, because they do not need to be transformed in external Transitions from the perspective of the actual TROPIC algorithm. For the colouring it is interesting to know the other side, but this is discussed in the next section. So if the assumption (Assumption II) is made that a one-to-one Relationship is never translated in

4. Transforming QVT Relations to TROPIC

an own Transition, the transformation becomes much easier. It can be executed directly without knowing any equality or semantical linking respectively. Only one-to-many complex types have to be checked for equality anymore which is a heavy improvement.

The next and only open issue is to discuss, if one-to-many Relations can be reduced. According to the QVT standard no obvious restriction of such Relations is discussed. Nevertheless, the usage of two or more one-to-many constellation leads to many-to-many situations – two 1:n-relationships mean $N \times M$ or three mean $N \times M \times L$ – the cross-products of the matches. These situations would, therefore, lead to a set of possible combinations that should be too high for the execution of QVT Relations and, therefore, be avoided by QVT engines (a warning should be presented or even better the execution should be denied). For this reason the visualization does not need to be able to translate more Relations than QVT engines can handle. So the assumption (Assumption III) is made that every complex type – including all Domains! – can only have one one-to-many related complex type as property. Nevertheless, it has to be realized that these three assumptions do not handle all problem situations. The third assumption leaves a gap that is described in the Listing 4.3. This issue is not present in the mainly discussed example in Figure 4.1.

Listing 4.3: Reusing one-to-many related properties

```
61
62 enforce Domain rel t:Table {
63     cols=cl:Column{
64         name=an,
65         type=pn
66     },
67     pk=pk:PKey{
68         cols=cl
69     }
70 }
```

There exists a one-to-many related property with the name `cols` which is transformed. The essential code element is placed after at the property `pk`. This element is a one-to-one related property (which is the only option according to the Assumption III), but it holds a one-to-many related property with the name `cols`. This is a situation that could lead to too many matches and confusing results. Hence, this constellation has to be forbidden as well. This means that Assumption III has to be extended. From every complex type at any place there exists only one path holding somewhere a one-to-many related property on the full length of the path. However, this path can hierarchically hold one-to-many related properties which is not explicitly disallowed. Coming back to the discussed example of above, it has to be mentioned that this example in reality fully fulfills this assumption, because it does only refer to a previously executed one-to-many related property. This is a certain exception that allows to execute the value of `pk` fully in the Transition of `cols`. There is actually no need for more hierarchy levels or other constellations violating the executability.

These three assumptions solve this problem much faster and computation-

friendlier than the discussed three step process. Furthermore, they do not leave any gap for special cases. So the further approach does not use this three step process anymore. Nevertheless, this analyzation procedure can be used for other interesting issues in the future. However, to be able to map the colours correctly and efficiently some more problems have to be solved. This is discussed in the next section presenting the problems beginning at generating colours and leading to colour-aware MetaToken productions.

Furthermore, the information handling also involves a hierarchical perspective on data sets encapsulated in Domains and its included and nested complex types.

4.6. Realization of Correspondences

To be able to colour the MetaTokens – which provides the only correspondence mechanism in TROPIC – it is obviously necessary to understand which Token (or its Place of origin) corresponds to which item of the other model. According to the previous section this cannot fully be defined, but most correspondences can be usefully identified.

Therefore, the colouring has to handle different types, cases and solutions. These differentiates are built upon the defined assumptions to link source and target side.

4.6.1. Transition “Owners”

Starting an execution the focus is typically set on the source side. A certain Domain of a specific Relation is going to be executed. This Domain is on one hierarchy level referring to some other levels that are enclosed. Primary the source model defines the colours that are used. Of course, there can occur items that are only used at the target model side, like conditions resulting from specifics of this certain model that does not correspond with the source model. Nevertheless, every Domain has to have one one-colored MetaToken which receives the complex type objects from a source Place or a TracePlace. For most cases there exists a MetaToken on target model side as well. So, how can this basic mapping be managed?

Each hierarchy level allows the storage of some data. By executing a source MetaToken a colour has to be taken. For the MetaToken that depends on the complex type that “owns” this Transition (Domains or one-to-many related complex type properties) this is called the *masteringColor* (in the application) that is stored in the specific hierarchy level. The *masteringColor* is the main colour of a Transition and, therefore, the colour for the basic element. Furthermore, this colour can be necessary for referencing to the previously created main *One-ColorMetaToken*. This way of storing the data is only possible, because of the previously defined assumptions that avoid parallel paths. Parallel paths would be hard to handle as discussed before, because it is not to hundred percent possible to link both model sides with each other. But the defined assumptions and the used hierarchy level storage enables the target side to consume a colour for a complex type. A second value is available for most other issues. This is called

metaLinkingColor and represents a certain alternative colour value for simultaneously linking a type with another type within the same Transition. This concept is presented in more depth in the next colouring issue explanations.

4.6.2. Relationships

The execution of properties can be split up in several different cases. The realization of correspondence between source and target objects is especially different for relationships of one-to-one or one-to-many nature. Hence, in this section the main differentiation is done according the multiplicity of the relationships. Beyond this fact, a determination of complex and primitive types is necessary.

One-to-one Relationships

The most essential variant is the one-to-one (1:1) related primitive type property. This only implies the usage of one TwoColorMetaToken that uses the masterLinkingColour of this hierarchy level as FromColor and another value as ToColor¹. This colour, of course, has to be unique and should be trackable for the other execution side.

Therefore, the colours could be stored in an ordered list, but this would imply that 1:1 related types could mix up their colours. To avoid the confusing mixture of complex and primitive types and to improve the probability of correct matches, they are stored separately. For example it could occur that one side has the condition `isPersistent=true` (like in Figure 4.1) whereas the other side only produces a new complex type called `type`. It would be confusing, if Tokens of `type` would get the colours of the Tokens from `isPersistent`.

So, in lieu thereof the type of variable for each Relation is retrieved. If a variable is found, meaning a two-sided relationship, the colour needs to be stored with the name of the variable. If a boolean, string, numerical or other constant value is found, a unique colour has to be requested by the ColorManager - the ColorManager is a construct provided by the TROPIC language implementation and is extended by own colouring mechanisms. This colour need not be stored at all, because there is no scenario of reuse known. A bigger problem is the colouring of complex-types properties that are one-to-one related – one-to-many relationships are unproblematic, because another Transition is produced and this complex type somehow “owns” it and, therefore, the knowledge of correspondences – structural equivalences – on this level is rather easy. Complex types always hold a variable and, therefore, also a variable name. However, the variables of source and target side are never the same. So if on the source side a complex one-to-one related property occurs, it can, but need not, reflect another type of the same structure on the target side. Hence, the colours of complex types cannot be stored and reused. There is the possibility of trying to calculate probabilities or to analyze the further usage of these variables (Where-, When-Clauses, nested variables matching on each side). For example a similar naming could reflect the wish to

¹Within the application the ToColor is often referred to the term innerColor, because it seems more intuitive from the graphical perspective. However, the logical statement of “from” and “to” seems useful as well

reuse the information, but in reality this is not be useful, because source and target side normally use different names. Therefore, at this point it would be desirable to have the equivalences analyzation component discussed in the section above. However, for the practical solution of this thesis, this seldom scenario is left out, because it can only lead to major execution problems, if a following Relation reuses the information from these variables. This is not typical for normal complex types that do not represent the owner of their Transition, but this can, of course, occur sometimes.

The execution of a complex type that is one-to-one related to its parent complex type, is similar to the execution of primitive types. Nevertheless, there are some special exceptions that have to be considered to be able to execute them in the context of a highly reusable system. The TwoColorMetaTokens are translated as for primitive types. However, another OneColorMetaToken has to be produced. A one-to-many related complex type can be seen as just a complex type like a Domain that is executed in the same Transition as the parent complex type. This means that the execution process of OneColoredPlace and OneColorMetaTokens can be expected by the method that executes the main features of a complex type. To be able to reuse this method some colour specific settings have to be done. To be able to combine other nested properties (except one-to-many relationships, because they “care” about themselves) the colouring of the masterLinkingColor and masterColor have to be changed temporarily for the execution of this property and all of its nested properties. However, after the execution is important that the previously used colours are utilised again to be able to transform all properties in the same manner that are placed on the same hierarchy level. So, before executing the child elements for a complex type the colours have to be stored temporarily on another place. For every child that is executed, the masterColor and the masterLinkingColour are set back to this original value. Then, if a one-to-one related complex type occurs they can be set to different values for the execution of this single property. What values should be stored for these colours? The innerColor (ToColor) of the TwoColorMetaToken that is produced similar to items for primitive types in one-to-one relationships are stored as both colours, From- and ToColor. This is needed, because the execution process refers to these colours to be able to set the FromColor for enclosed properties. This leads to properties that are translated in the same Transition, but obviously show their dependency to their certain direct parent type. The main type (the type that “owns” the Transition) of the Transition stays untouched, because it is needed for further executions.

One-to-many Relationships

In contrast to one-to-one, one-to-many (1:n) relationships are more difficult as already discussed. This continues with the realization of correspondences (colouring), because more colourings have to be clarified. Furthermore, the colour data for one of such executions has to be present for both parent and child Transition. This means that the conditions placed in the parent Transition have to be set to the same colours like the classical OneColorMetaTokens (and TwoColorMetaTokens) as described in the one-to-one relationship section. The latter execution is logically and programmatically the same as discussed before. Furthermore,

4. Transforming QVT Relations to TROPIC

the MetaToken in the new Transition (higher hierarchy level) receiving information from the parent Transition (lower hierarchy level) has to be set. This is done similar to the execution of Where- and When-Clauses in the combination of TracePlaces. Again the MetaColor for the receiving trace information is reused from the masterColor that is used in the lower hierarchy level and represents the masterLinkingColor of this level. This is done to be able to combine all referred properties from this colour. Then a differentiation of primitive and complex types has to be made again. Both need a TwoColorMetaToken that links the receiving colour to their own value. However, a complex type does also need an own One-ColorMetaToken with its own colour value. This is again done in the execution of the basic proceedings for a complex type initiated at the current state.

Therefore, the execution and colouring of one-to-many related properties with complex types is a recursive process that needs more pieces of information to handle the linking process of the two related complex types.

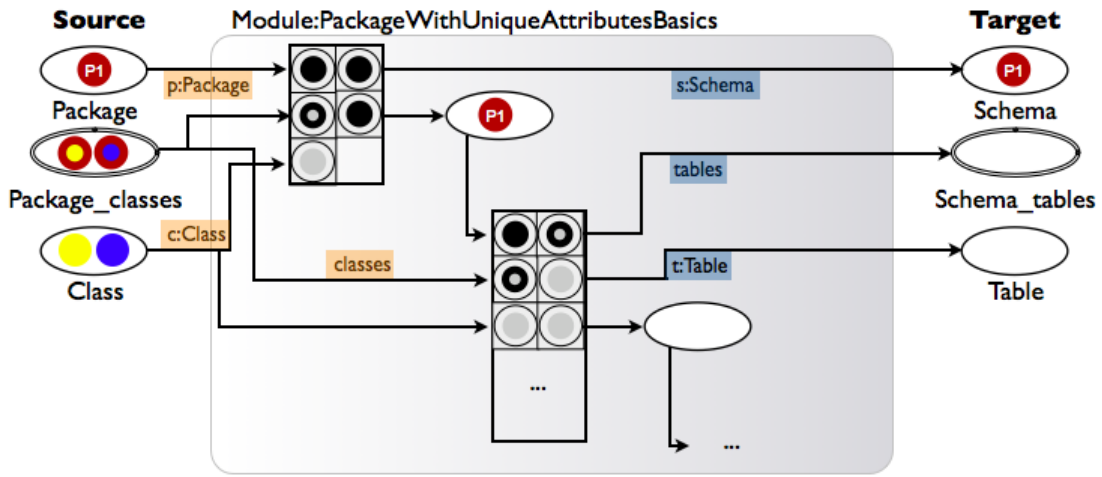


Figure 4.8.: One to many relationship example

An example is placed in Figure 4.8 which represents a subview of Figure 4.1. A single **Package** instance can hold n instances of **Class**. Hence, the TwoColored-Place **Package_classes** and the TwoColorMetaToken exist that combine both involved types. The colouring of the MetaTokens which is the most essential can be seen in the first Transition (from the top). The grey meta colour of **Class** is the inner colour of the TwoColorMetaToken. The border represents the “origin” which is in this example black of the type **Package**.

4.6.3. Referred Relations

For Relations that are called by a calling Relation via Where-Clauses, but also for top-level Relations that imply some pre-conditions that have to be executed before (When-Clauses), a special treatment has to be done. At this point it seems notable that TracePlace executions can only be source transformation executions, because the consumption of information is a source side task and it seems to be avoidable that this linking is done twice.

Instead of producing an own MetaToken in the Transition a value is consumed from another Relation or Transition respectively. This colour received is obviously

a colour value that need not represent the value that is important for the called Relation in the sense of a typical main type execution, because the received Tokens need not be transmitted to the target Places. Therefore, the colouring of this scenario needs some further assistance. It seems again notable that TracePlace consumers resulting from a Where-/When-Clause use the transmitted information as main type information which means that the colour of its MetaToken represents the main information to which all other properties refer to. In all other cases another main type and main colour respectively is available. In one-to-many relationships the referred complex property is the main type and, therefore, its colour matters most. However, the linking between passed and main type value has to be prepared as well.

First of all, the main colour (`masteringColor`) of the calling Relation has to be retrieved. This is done by checking the parent Relation (this is only possible using the information handling process discussed in the section about the realization of this thesis). Nevertheless, it cannot be so easy to just take the mastering colour of any hierarchy level and to create a linking from this Transition to the main type of the new Transition. In the first considerations it could seem rather necessary to get the last (highest counter) hierarchy level that has been added to the parent Relation. However, it has to arise the question why it should be the last hierarchy level. This question is quite more interesting than it seems at first glance. For example in Figure 4.1 transforming the type **Class** that holds some attributes (`attributes`). **Attributes** is a one-to-many relationship resulting in an own hierarchy level. However, it is the intention of this Relation to pass on the variable of the type **Class**. For this purpose it seems to be wrongly placed to use the last hierarchy level. However, this usage can be quite advantageous, because this level of information provides something more useful than the first or any other level. It is a guarantee that only Tokens are passed on fulfilling all conditions set in the whole Domain (or Relation respectively) execution. Going back to the example of a **Class** holding some one-to-many related attributes, this Domain statement means that a class that is passed on has to have these certain pieces of information represented by the Tokens of these attributes. However, for the production of colour-aware transformations it seems wishful to combine the advantages of using the last hierarchy level or the very specific one transforming the passed type. Respecting identities and flows the correct colouring is very essential. On the one hand it has to be determined that only Tokens are passed for which all conditions are fulfilled – like in the usage of the last hierarchy level – and on the other hand it is necessary to pass the correct colour from the correct position to the next Relation. The latter issue is important to allow intuitive information handling. Hence, the hierarchy level has to be chosen carefully. The Transition transforming an element of the type **Class** should directly pass its value to any receiving Relations using exactly the same colours. For example a **Class** could hold a child type. If the information (and, therefore, also colour) of the child is passed on, this results in different used colours for the same instance of **Class**. This has to be avoided to allow intuitive debugging and to represent useful relationships. See the following section for a solution for avoiding improper partial states that can result from the information passing from a Transition that is not the **last** Transition within one Relation. In the context of the implementation in

4. Transforming QVT Relations to TROPIC

this thesis, the information is always directly passed on and, therefore, the later on presented modifications have to be accomplished.

From this chosen level the mastering colour can be received and stored as `masterLinkingColor` in the new hierarchy level of the child Relation (the actually executed Relation). This colour needs to be used to produce a `MetaToken` for a new `OutPlacement` in the parent Transition, because it is important to pass the mastering information on to the next Relation. The same colour is used for the `MetaToken` receiving information from the set `TracePlace`. This is no need, but a comfortable way of visualizing this. It is no necessity, because the colouring only needs to be loyal within one single Transition, meaning that some value that is received is send to the target in a certain combination. Hence, they could also be mixed and rearranged in the target model. However, this is not really used effectively in this thesis, because no real execution scenario has been found for it. Going back to the colouring problem of `TracePlaces`, the colouring of the main type is realized setting or getting the `masterColour` of the child Relation. For a source side execution this colour has to be generated, while for target side it can be picked from the stored value. The linking between the `MetaToken` only holding the `masterColour` (main type) and the received information over the `TracePlaces` is done by creating `TwoColorMetaTokens` with `FromColor` (border) of `masterLinkingColor` and `ToColor` (innerColor) that holds the `masterColour`. Furthermore, all other properties can be linked afterwards using this colour scheme.

4.7. Complexity and Effort

The possibility of cross-products and the need of computing permutations of all Domains for each Relation showed the importance of algorithmic sensitiveness for transforming QVT code to Transformation Nets. Hence, this section tries to give an idea how the complexity can rise by using different constellations of QVT Codes. This thought is the basis for developing the practical graphical debugger solution, because the restrictions of QVT engines and the possibility of visualization in Petri Nets has to be considered in every detail to be able to produce a useful solution.

Therefore, it seems a necessity to go back to the algorithmic processing of Relations. A top Relation can be seen as root of a computation tree holding other referred Relations as child nodes – please, keep in mind that the $\text{top}(N)$ Relations with several dependencies through postconditions (When-Clauses) have to be included with special care. Computing a flow from one top Relation to all of its deepest child nodes (Relations) evolves a process similar to depth-first-search (DFS) algorithm. However, the When-Clauses can interrupt this computation process, what could lead to the skipping of tree parts. This does not change the computation effort intensively from DFS-search for each top Relation. The space complexity of a depth-first search is known to be

$$O(b * m)$$

and its time complexity is considered to be

$$O(b^m)$$

(O-Notation of absolute value of vertex and edge counts) which is the worst-case complexity (upper bound) [27]. The parameter **b** is the “branching factor or maximum number of successors of any node” and **m** represents the “maximum length of any path in the state space” [27]. The involvement of several different top-level Relations – leaving out the problem of missing preconditions – would rather lead to a definition of different maximum tree heights. An index called **i** is needed to differentiate between the different depth of Relations. However, the execution path that is needed to know all information data is much more complex. This clearly shows that the complexity effort has to be calculated differently, but very simplified to show its importance for this thesis.

Each Relation that is executed (once or several times) holds preferable two Domains that are of complex nature. It could also include several more Domains, but this is not used in this implementation according to the difficulty of finding correspondences within each pairwise combination of Domains. A Domain pattern can hold several primitive types, complex types that are in one-to-one or one-to-many relationship to it. A one-to-one related primitive type does only imply the transformation of one value. A complex type is much more challenging. Just remember the infinity of other properties within a complex type (broadening) including the basic Domain types. Furthermore, the infinitely deepening through nesting of complex types (deepening) is a computation problem as already discussed before. So, the usage of complex types obviously leads to an infinitely set in the worst case. However, the situation for one-to-many related properties is even worse (the infinity of the worst case cannot become worse, however the average computation can worsen) compared to its one-to-one counterparts. One-to-many related primitive type properties cannot hold infinitely often the same type (broadening) – see Figure 4.6. For example it can hold a set of **names** as simple string values. This broadening is an additional broadening to the previously discussed situation. The one-to-many related properties are similar to other complex types, but involve like other 1:n relationships the further broadening for each used complex type. Therefore, the complexity of such a type could really get amazing. Constructing a calculation from this idea the computation effort for each Domain of each Relation could be formulated similar to the following formula 4.1. In this calculation the computation effort for each processing step is considered to be equally at the constant value of 1. This is somehow comparable with the number of instructions needed to compute a certain transformation visualization.

$$\begin{aligned}
 CC = & \sum_{i=1}^R \sum_{j=1}^D (1 + \sum_{p=1}^{PP_{ij}} (NP_{ijp} * B_{ijp}) + \sum_{p=1}^{PC_{ij}} (NC_{ijp} * B_{ijp} + \\
 & + \sum_{k=1}^{K_{ijp}} (complexTypesOfEnclosedProperties)) + \dots + \dots)
 \end{aligned} \tag{4.1}$$

The used indices and abbreviations are explained in the Table 4.1.

This formula only considers some steps from a complex type object deepening to its child Relations. This algorithm of involved mappings grows for each

4. Transforming QVT Relations to TROPIC

| Abbreviation | Description |
|-------------------|---|
| CC | Computing complexity |
| i | Relation index |
| j | Domain index |
| p | Root level properties index |
| k | Enclosed properties index |
| PP _{ij} | Number of primitive properties of a certain Domain of a certain Relation |
| PC _{ij} | Number of complex properties of a certain Domain of a certain Relation |
| NP _{iju} | Each primitive property of a parent complex type (Domain) of a certain Relation |
| NC _{iju} | Each complex property of a parent complex type (Domain) of a certain Relation |
| B _{iju} | The boundary (broadening) of a specific property (one-to-many or one-to-one) |

Table 4.1.: Description of the used abbreviations

attempt of deepening, because for each parent type many child types can be used, but for every child type only one parent type can exist. Simplifying the stated formula, would lead to a deeply nested constellation. The formula stops at the computation of the child elements of related complex types. The calculation would be similar, but the deeper the calculation becomes, the more indicators are needed to track the actual state of execution. Hence, the brackets holding **complexity of enclosed properties** would get deeper and deeper and this is clearly out of the scope of this thesis. Furthermore, it is not the intention of this formula to fully calculate the complexity of such a graph constellation, but to give an idea, how the effort grows with each node (especially nodes with more than one outgoing edges).

To be able to consider a more realistic effort calculation the space needed to store and the time needed to compute the certain instructions would have to be weighted. Besides this necessity, the effort can also be viewed at in different perspectives. First, as already mentioned, the theoretical calculation of time and space complexity seems appropriate. This can be seen as the effort to produce such items. Second, each element enforces other elements that are needed on the Transformation Net as well. For example a MetaToken typically also involves the usage of an In-/OutPlacement and an Arc. This could also lead to a rethinking about complexity computation based on dependencies. Third, the visual representation can be seen as the used resource that matters most. There is only a screen with certain – typically too low – screen resolution that has to visualize a numerousness of elements. Therefore, a Module or Transition could be more costly than the production of a MetaToken.

In all usage variants the determination of complex/primitive types in one-to-one or one-to-many relationships matters most. To see the relationships of the different types in an execution process, the set of all contained types for a single complex type (without children) can be visualized as mathematical set of data.

See Figure 4.9 for the graphical representation of this idea. The outer boarder marked with **n** is the set of all properties contained in this complex type. In the best case – for the execution complexity – this set is empty as all others have to be empty as well as consequence. In the worst case it holds an infinite amount of other types. In this set two other sets are contained. One is called **NC** meaning all complex types. The other one is the inversion of this set and, therefore, **1-NC** meaning all primitive types. The one-to-many related properties **1:m** (**NCM** for complex types and **NPM** for primitive types) are part of both of the previously described sets. Naturally, all depend to the set **N** of all properties.

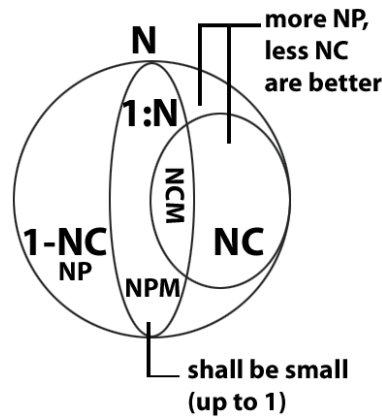


Figure 4.9.: Determination of different types contained in a complex type container

Nevertheless, this should only give an idea about the need of some restrictions. For sure many problems were not considered in this short review of complexity and effort respectively. For example it has to be left out, that every step includes searching through the whole tree elements, internal data structures, types or nested types.

The effort for computing such transformations is not easily (monetary) valuable. However, the complexity of the involved steps and the effort to compute a single execution (a simple statement leading to a simple output value) and the related amount of writing out the results, would lead to a certain effort value. In this thesis only the execution and serialization effort is valued. Details about these efforts are presented at the evaluation of the graphical debugger. Furthermore, it seems notable that although from outside it might appear differently, the full process is deterministic and could be pre-estimated before, because the used algorithms follow strict processing rules.

4. *Transforming QVT Relations to TROPIC*

5. Supporting Advanced Features of QVT Relations

In the previous section the transformation of QVT-R to TROPIC elements has been discussed. However, there exist advanced features that have not been mentioned. In this section we will discuss meta-class inheritances and its mapping to TROPIC. Furthermore, solutions for incremental changes are identified.

5.1. Inheritance

Another often used feature of MOF models is inheritance of classes. If a class is a subclass of another class (parent class), then all of its instances are indirectly instances of this parent class as well. This fact has not been visualized in the TROPIC Net so far. For each used type – representing a class in the model – a Place is created as discussed in the previous section. If there is a wish to visualize unused superclasses, the Places for these types have to be created before creating any Tokens, but after visualizing the transformation rules.

The model loading feature of TROPIC creates for every instance a Token for its direct class. This has to be enhanced to create duplicate Tokens for all indirect classes it is an instance of as well. This allows the addressing of direct and indirect instances in different transformation rules. However, as all duplicate Tokens represent only one object (instance of a certain class), a functionality is necessary that allows the merging of Tokens in the target Module. This is a necessity for a consistent representation and has to take place at the end of the transformation.

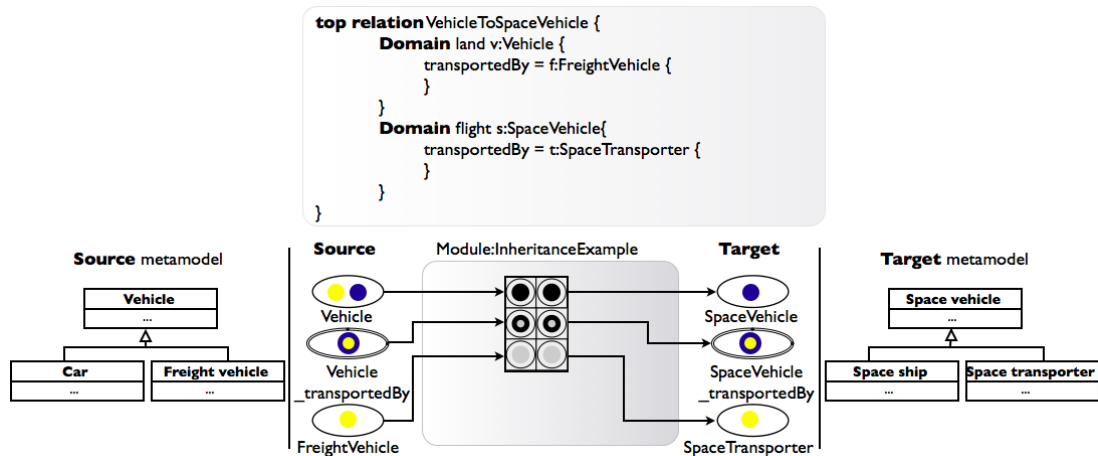


Figure 5.1.: A TROPIC example with metamodel inheritance

5. Supporting Advanced Features of QVT Relations

The usage of subclass and its parent class is shown in the example of Figure 5.1. Instances of `Vehicle` are only transformed, if they have been transported by a `FreightVehicle`. Therefore, the instance representing the yellow Token is transformed as by-product of the transformation of the blue Token as it does not satisfy this condition itself. If it is desired to express that all `FreightVehicle` instances are instances of `Vehicle` as well, a duplication functionality could be called after creating this Transformation Net.

5.2. Incremental Changes

Incremental changes for models are modifications of existing models by adding, deleting and updating model elements. For TROPIC nets this could mean the adding of another Token in a Place whereas the rest of the net stays the same. Such incremental changes should not lead to the production of fully independent model version as this could harm the intended consistency over model versions. Therefore, synchronization approaches can be used to propagate modifications back to the previous model version or even to related models. This means instead of reproducing models, the changes have to be analyzed and integrated in the other models.

5.2.1. Introduction

Incremental changes can be interpreted differently on several (granularity) levels for TROPIC nets, because there is no common understanding available that would define incremental changes in the context of such nets.

Incremental changes only imply that models are modified to improve or enhance the models instead of providing the fully correct or intended model at once. Moreover, incremental changes can be caused by repeatedly modifying models to achieve a certain intended result. Nevertheless, models could also be shared by several developers who try to place their model information (e.g., test data) in the model. In contrast to the repeatedly modifying elements, these parallel pieces of information stored in the model need not directly correlate. However, the handling of both types in this thesis is the same. In this thesis incremental changes are addressed by model synchronization approaches. Synchronization allows the propagating of model changes to other model versions. This allows the incremental modification of models without reproducing the whole model itself, but only propagating the identified changes.

5.2.2. Model Synchronization

Model synchronization allows the consistent merging of model states. Models can be seen as MOF models (such as the input and output models for QVT-R) in this thesis. The QVT-R code as well is parsed to a model representation before it is visualized. The transformation net in TROPIC is a model based on the TROPIC metamodel.

Model synchronization for model transformations is described with the help of the example of Figure 5.2. A model is created (`Model A0`) and transformed to its

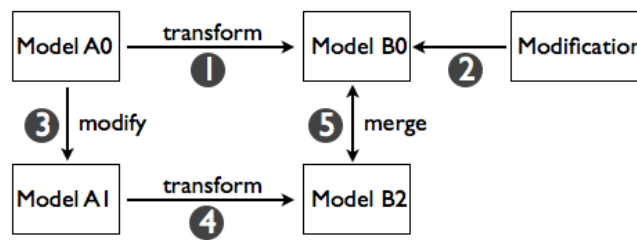


Figure 5.2.: Model synchronization for model transformations

target model representation (**Model B0**) in step 1. Afterwards another interaction (step 2) with the resulting model takes place by modifying it. These modifications need to be kept for the future. In step 3 the original source model is modified (**Model A1**) and afterwards (step 4) the model transformation is rerun. Therefore, an output model (**Model B1**) is created that does not contain the modifications added to **Model B0**. Therefore, a merging process (step 5) has to take place that propagates the changes from **Model B0** to **Model B1**. This process is the model synchronization.

Variants of Model Synchronization

In this thesis model synchronizations are used in three different variants: First, the synchronization of the input and output model; Second, the synchronization of QVT-R and its TROPIC representation; Third, the synchronization of the execution trace. All three variants are sketched in Figure 5.3 and marked with dotted lines and numbers.

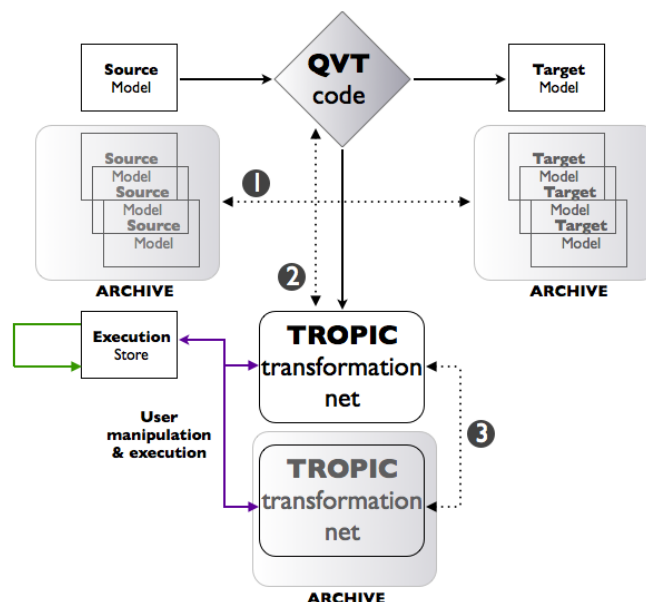


Figure 5.3.: Different variants of model synchronization in Grade

Model synchronizations can be achieved by archiving previous versions of models. If a new model is created from a certain input, it could be built to a history

5. Supporting Advanced Features of QVT Relations

area first. Then the result could be compared with the previous applicable result and then be merged. Only particular changes are considered in the merge process. The merged model is then used as actual model.

Input / Output Model Synchronization There is the possibility of synchronizing the input and output models (1 in Figure 5.3). Using a certain QVT Code a source model is transformed to a target model. By changing the input model at some point, the target model should not be newly created. Only the changes should be propagated to the target model.

Net Synchronization The created TROPIC net can be synchronized with its QVT-R representation (2 in Figure 5.3). Every change in the QVT mappings should be reflected to the net without reproducing the full net. Typically the user adds new Tokens to be able to test certain Transitions. Such modifications should be keepable to improve the effectivity and efficiency of debugging with Grade.

Execution Trace Synchronization The execution trace from one model to a certain result using a transformation net can be used (3 in Figure 5.3). As the QVT-R code may change dramatically over versions, the pure synchronization of models might not be sufficient. This results from the problem that transformation paths from source to target may change intensively. For example, if a certain Token results from firing particular Transitions, this could be somehow different to the result that could be achieved in the TROPIC Net based on the new QVT-R code. This change of behaviour needs to be propagated over the whole transformation path. For such situations the execution trace synchronization is useful.

Instead of synchronizing models, the user interactions with the TROPIC net are tracked. If a user fires several Transition, he or she typically tries to identify particular problem areas (origins). After realizing that a certain erroneous behaviour occurs, the QVT-R code is changed. Afterwards a new TROPIC net is created. This Net should be able to be set in the same condition as before to see, if the result can be achieved that has been expected. Therefore, this approaches allows the replaying of the execution trace by firing the Transitions in the same manner as in the previous debugging session. This can only be done as long a firing is possible. If firing a Transition is not possible anymore the replayed execution path has to be stopped and the user has to be shown a prompt that informs him or her about this fact. Such execution traces should be keepable over versions allowing naming and selection.

A drawback for this approach is the fact that TROPIC uses names (or sometimes not even names) for identification of elements. So, if an element such as a Transition is renamed, it cannot be found anymore. The firing of the execution trace has to stop at this point.

Adopting Model Synchronization in Grade

The synchronization use case being adopted to the prototypical implementation focuses on the synchronization of Tokens. The Token synchronization is a subset of the TROPIC Net synchronization use case. The Tokens are used as the only hold one particular value – their colour. This colour is sufficient to identify them within a Place. Their synchronization is, moreover, very valuable as modifying, adding or deleting Tokens is a very commonly used debugging scenario. For the synchronization of Tokens in Grade three particular synchronization needs could be identified: First, it should allow the possibility of synchronizing the Tokens being placed in the source Module. Second, the Tokens being placed in the target Module can be necessary as well. Third, the information stored in all intermediary Places (TracePlaces) in other Modules should be synchronized as well. As these variants are realizable, they are integrated in Grade.

5.3. Summary

In this section some advanced model transformation topics have been discussed. In particular the areas of class inheritance and incremental changes have been discussed. On the one hand, incremental changes have been addressed by model synchronization. Several use cases for model transformations have been identified in Grade. On the other hand, class inheritance has been addressed by duplicating Tokens in the source Module at the beginning of the transformation, and merging the Tokens in the target Module at the end of the transformation.

5. *Supporting Advanced Features of QVT Relations*

6. Realization

This section concentrates on technical, programmatic and realization-specific issues. Therefore, the used IDE (Integrated Developer Environment) to implement this graphical debugger as well as the used components are introduced. Then the actual accomplishment of the execution process is presented. All topics described in this section fully correspond to the conceptual approach discussed in previous sections.

The prototype called “Grade” can be considered as prototypical implementation showing a specific way of visualizing QVT-R Code. However, the user interface is not intended to be designed for professional usage up to now.

6.1. Environments

Naturally, not all used components or tools are built from scratch for the application presented in this thesis. There is an intensive reuse of IDE functionalities and external components that provide the opportunity to develop in a modern manner and to focus on open issues addressed by this thesis.

6.1.1. Eclipse and its Plugins

Grade is developed in the open-source IDE Eclipse using the Eclipse Modeling Framework (EMF) [28] to create the models on which the Transformation Nets are based. To be able to use the features for modeling and reading models of EMF, Java 1.6¹ is used to create this application. A TROPIC plugin (for Eclipse EMF) connects this implementation with the algorithm used in the TROPIC project.

For the purpose of this project, an Eclipse 3.4 Ganymede Modeling Edition configuration is used. This specific version provides the basic Eclipse 3.4 functionality combined with a set of plugins that fulfill important tasks in the area of modeling and model transformations. This configuration contains the Eclipse Plugins “Eclipse Modeling Framework ” (EMF) in Version 2.4.2, “Kermeta” (Version 1.2) and “Graphical Modeling Framework” (Version 1.1 and Tolling version 2.1). These three plugins – as some related – are needed to view, run, read, write and use the models written in Ecore format. All parsed QVT-R codes are transformed to Ecore models that are evaluated and visualized. Furthermore, the TROPIC plugin needs these Eclipse plugins as well.

The most important component used in this practical realization of this thesis is the TROPIC plugin. This is needed to visualize the information with the algorithm specified by TROPIC. Basic intersection point for this graphical debugger

¹<http://java.sun.com/javase/6/> - last accessed: November 27 2009

and the plugin is the construction of a Petri Net using the factory specified in the plugin. The graphical notation provided by the editor is similar to the previously pointed images. However, there is some derivation resulting from the graphical placement of elements on the Transformation Net as there is still room for improvement concerning automatic layout algorithms.

6.1.2. QVT-R Parser

To transform a textual QVT Relations code formulated as before to an EMF QVT model, a free QVT-R parser available on SourceForge has been used. The parsed QVT-R Relations are stored in an EMF model conforming to the QVT metamodel. Figure 6.1 shows an extract of the metamodel for QVT Relations used in this QVT Parser.

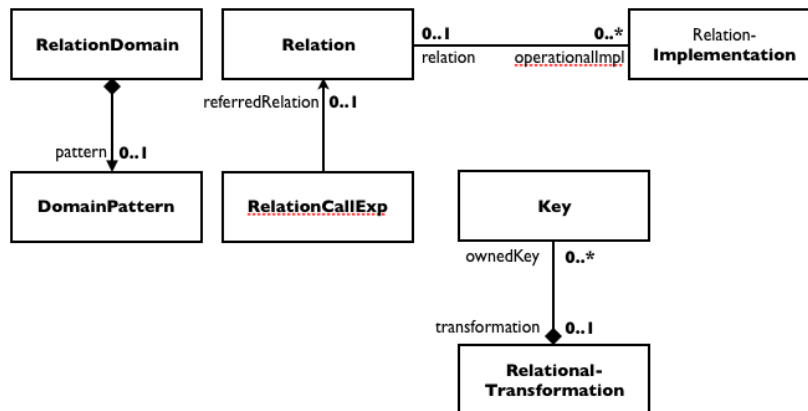


Figure 6.1.: Metamodel for QVT Relations used in the QVT Parser

In this metamodel a more specific representation can be seen that reflects the model (as XML-file) in which the identified QVT-R elements are stored by the parser. This model is used for the computation in Grade later on. As from this metamodel settings like boundaries have be derivable, finer granular differentiations are necessary than in the simplified metamodel shown in the sections before. For example a **RelationDomain** is the same as a stored Domain. The included pattern - the complex type that refers to the Domain - is called **DomainPattern**. Furthermore, many differentiation like top-level or non-top-level can be stored using normal boolean attribute values. Relations that can be called using Where-Clauses and similar mechanisms do not directly call the Relation, but hold an instance of **RelationCallExp** that refers to the Relation itself. These changes in naming and the finer differentiation of types does not change the overall mechanisms of mapping QVT-R to TROPIC as they have been described in the sections before, but they are necessary for retrieving the information in the code and are, therefore, mentioned here.

Generally, this parser helps to classify and read the structures in a QVT file, but it does not help to understand, sort, interpret and visualize the data. Hence, this application is used to prepare a common basis on which analysis is undertaken.

¹<http://sourceforge.net/projects/qvtparser/> - last accessed: November 27 2009

Moreover, the parser cannot only analyze QVT-R, but also Core and Operational rules which are effectively not used in this approach.

6.2. Implementation Specifics

As this thesis propagates the Grade prototype many details concerning the implementation solution exist that are presented in this section by addressing the following areas: (1) Many data types are used in Grade that are important for the execution processing; (2) The encapsulation of information is an issue resulting from the graph-transformation approach involving a reduction of accessible data per node; (3) The usage of patterns and strategies leads to a specific class structure which needs to be introduced;

6.2.1. Data Types

This section discusses the usages of important data types resulting from the QVT parser. Those types have a storage and sometimes management functionality. The entities resulting from the QVT parser are metamodel conform – e.g., `Relation`, `Domain` and `DomainPattern`. In this section the `Predicates` with `RelationCalls` being used for identifying the execution ordering of Relations, the `OclExpression` accomplishing value assignments, and the particular representation of complex (`ObjectTemplateExp`) and primitive types (`PrimitiveLiteralExp`) in the application are discussed. Therefore, these types are essential for the visualization of QVT-R in TROPIC.

Predicate and RelationCall

The `PredicateImpl` is the information provider in Where- and When-Clauses. Such Clauses consist of several Predicates that can call other Relations `RelationCallExp` with a value set (variables).

OclExpression

The `OclExpression` - or `OclExpression` respectively – is the parent type of all types (primitive and complex). This has not been mentioned so far, but, of course, the naming results from the fact that each condition or mapping is an expression specified in OCL constraints.

ObjectTemplateExp and PrimitiveLiteralExp As the `ObjectTemplateExpImpl` – implementing `ObjectTemplateExp` – is used for storing complex types like Domain patterns, it is essential for the Grade computation. The primitive types extend the `PrimitiveLiteralExp` class – e.g., types for Boolean (`BooleanLiteralExp`), String (`StringLiteralExp`), Numeric (`NumericLiteralExp`), Variable (`VariableExp`) exist. The `VariableExp` allows source-target correspondences and value passings to other Relation, and are used for complex types as well.

6.2.2. Information Encapsulation

A model is a worthy way for encapsulating data for a certain problem scenario. However, the model produced by the used QVT parser cannot provide all pieces of information needed to be able to handle the full course of transforming QVT-R to its Transformation Net representation. Hence, by starting the execution of a certain Relation, the received parsed Relation needs to know which Domains it handles. Furthermore, every Relation is translated into an own module. This module is used throughout the execution of all contained types (Domains, properties). Therefore, it has to store the module. Additionally, TracePlaces have to be remembered to be able to access them later on to reuse them or change their information content. To be able to trace this flow from one Relation to another – which is not possible using the passed Relations - the parent Relation has to be known and stored additionally. According to the colouring issues discussed in the concept of this thesis, some sets of data have to be stored for each hierarchy level of a Relation. A hierarchy level is a set of data transformed to one Transition. This implies that the usage of a one-to-many related property results in a new hierarchy level with its own Transition. The information handled in a certain level is basically the colouring of the main type – as described in the approach before –, a linking colour (linking to the parent type), a colour list for variable mappings and a sorted list of colourings for complex types that are not the main type of the Transition. The separation of complex types is undertaken to avoid the mistake of mixing the colours of a primitive and complex type – this could confuse the developer quite a lot. Going back to the original problem of a certain information need for Relations in the phase of executing them, it is obviously necessary to put them in a container. In this thesis this container is called **RelationItem** and its counterpart for Domains is called **DomainItem**. The latter container is used for each Domain of a Relation. Therefore, a **RelationItem** can contain several **DomainItems**. **DomainItems** itself are able to refer to the **RelationItem** they belong to and, furthermore, hold the information of Domains. At this point it seems notable that there does not exist something like a **PropertyItem**, because the execution is too similar to Domains and, therefore, is maintained using other storage mechanisms. Together all these containers allow the “jumping” from one item to another one without parsing the whole model. This form of packing the original information in a construct holding additional information is a concept called “container” [4]. This usage also implies a certain management of the encapsulated data using the container instead of directly accessing the Relations or Domains. This helps to include often needed processes or queries in a clean and easily accessible way.

This way of storing the data is helpful in navigating from one item to another, but it does not fully solve the issue of reusing code parts. The process of transforming data from QVT code to a Petri Net involves many repeatedly used TROPIC Net production Nets (such as the creation of Places and their registration in the system) that can be sourced to an external factory or similar construct. To be able to execute the correct information item at the correct time, there is a need of remembering what special item is to be executed at this moment. Hence, a **GradeExecutionCard** is introduced to track the actually transformed complex type. Moreover, this card helps to manage the actual objects and to refer to

related and previously executed items. Therefore, this flexibility is interesting for executing a certain method, because it does not need to know what it actually has to transform and it need not be specified by each method call in which context the execution should take place. Hence, the `GradeExecutionCard` can be seen as context provider for the execution process.

Another momentous advantage of using the `GradeExecutionCard` structure is the easiness adding new pieces of context information in the future without modifying any parameter passing. This results in a beneficial advancement of the overall software quality of the graphical debugger by raising the maintainability of the source code.

6.2.3. Class Structure

The basic structure of the application is split up in three sections. (1) Several libraries like TROPIC are used. They are only added to the project and available for usage. (2) The `QVTParser` and its classes are integrated in own packages of the project. The structure of this parser is left unchanged and is, therefore, easily updatable or replaceable. It is, furthermore, handled similar as an external component being only accessed at some known interfaces to decrease the dependency on external software modules. (3) The last component is the code of the debugger itself.

The internal debugger structuring is again split up in several categories. There are factories providing access functionalities to the TROPIC plugin. The executors provide the main transformation execution. Furthermore, several supporting categories exist to store and handle the actual state.

Factories

In this application several factory methods [10] – abbreviated with the term factories – are used mainly to decouple the Grade implementation from external components – or to reduce the coupling to some extent. Hence, the creation of Places, Arcs and Colors, as well as the loading of model information is delegated using a factory.

The TROPIC plugin provides a `ColorManager` that is able to provide unique colours throughout the execution of the debugger. This `ColorManager`, however, does not handle some application specific semantics represented in the colouring of Tokens. For example it could be wishful to provide a mechanism that automatically tries to use the same colour for the same types in all Relations - if this is possible according to some restrictions. Therefore, a factory is placed between the `ColorManager` and the application code. It tries to remember the colours and transforms the request to a correct delegation to the `ColorManager`. The advantages of placing this code in a factory are the exchangeability and modifiability of the `ColorManager` at one single place, as well as the reduction of the amount of code necessary to introduce this logic in every method using the `ColorManager`.

Moreover, the loading of model information is again a task that is delegated to an external plugin. To be more precise TROPIC fulfills this task. The model loading process requires the correct instantiation of the related class using correct parameter values. Therefore, this is placed in an own factory as well.

Executors

The executors provide the main execution work of the Grade application. They are split up into **RelationExecutor**, **DomainExecutor** and **TracePlaceExecutor**. The main focus of these classes is the handling of the logical flow of the application. Therefore, the information storage is not done in this executors – this, therefore a similar concept to the action/executor pattern².

The **RelationExecutor** is able to construct the basic flow of Relations and is, furthermore, responsible for delegating the transformation work to related executors. This means that the **RelationExecutor** checks preconditions – if the Relation is able to be executed – and then retrieves the Domains from the Relation. The execution of the Domains is delegated to the **DomainExecutor**. After executing all Domains the execution of all referred Relations can start.

This **DomainExecutor** is the component that produces nearly all elements that are added to the Transformation Net in the whole application. This results from the strategic positioning of Domains in the model transformation process. At the point of executing a Domain, the application should know the path it took to reach this Domain (Relation, parent Relation, ..), the Domain data and all referred properties. Therefore, a **PropertyExecutor** could only provide the possibility of pushing some code in other classes to improve readability and maintainability. However, this does not represent the meaning of the used “Executors” in this thesis that prepare the orchestration of the course of actions.

Another Executor is used for TracePlaces to be able to bundle the logics of the creation at a single point. The assignment of linking two Relations or types are done at several different points in the application. The logical execution is similar in all situations. Furthermore, the flow between these items is essential for the basic information flow within the produced Transformation Net.

In addition, it is possible to see a top-down weakening from **RelationExecutor**, **DomainExecutor** to **TracePlaceExecutor**. The Relations are central elements in all transformations and are, therefore, the corpus of the application. The organs providing the real functionality, however, are placed in the **DomainExecutor** class. The **TracePlaceExecutor** and the related factories can be seen as solution for not needing to provide all functionality at one point at every time.

Information

Naturally, the execution process needs data objects supporting the course of execution. For the storage of the actual state and for being able to reuse methods as often as possible, the execution state needs to be present. This state is stored in the **GradeExecutionCard** holding the actual object that is transformed and the context information. For example to transform a certain property it is a necessity to be able to access the Domain and Relation information. Furthermore, the **GradeExecutionCard** does not only provide easier storage and passing of information, it also helps to differentiate between different execution cases. The logics for this differentiation is bundled at this single place to be able to extend or modify it in the future.

²<http://msdn.microsoft.com/en-us/library/cc984279.aspx> - last accessed: November 27 2009

The already discussed `RelationItem` and `DomainItem` are also classified in this category, because they allow the encapsulation of relevant information (meta information or related pieces of information) for Relations and Domains respectively.

Furthermore, the execution process has to handle a handful of settings, like model names, file endings, temporary storage files or the enabling/disabling of some application parts. Normally such values are passed in the call of the application or are specified in a certain configuration file – typically a property or “eXtensible Markup Language” (XML) file. However, the information received at this point is not only relevant for the class representing the entry point, it is, naturally, also a necessity to be able to access these values in many (all) other classes. Hence, this configuration is managed in another class that returns the values requested by classes. Again, this approach enables the opportunity to exchange the storage format and handling at a single class without having the need of changing several other files that require this information as well. In the first release a relatively simply property file is used that stores key-value pairs in a text file that can be loaded in Java Hashtables. It is even used to maintain the staging process in Grade – it therefore always the configuration whether or not a new QVT-R model from a source code should be built, the synchronizer should be called, and model information should be loaded to the resulting TROPIC Net.

Another important information encapsulation is found in the `HierarchyLevel` class. A hierarchy encapsulates related TROPIC elements according the used input and output Metamodel – as described in the sections before. This class does not handle the creation and management of hierarchies, because they depend on a specific Relation (or `RelationItem`) that is, therefore, able to overtake this issue, but to encapsulate the information relevant for a single hierarchy level – this class can then be used in Relations for this purpose. For example the correct colouring is an issue that requires some detailed information about the hierarchy level. It is necessary to know the colour of the main type (`MasterColor`) and the colour – if existent – of the parent type. Furthermore, certain constraints regarding the colour management, the call of the factory responsible for creating colours, or the decision whether to create or reuse a colour, rely on the data stored for each hierarchy level. Some logics for such decision can be even moved to the hierarchy level to reduce the amount of misuses.

Moreover, the `RelationStack` class is used to store the Relations classified by Grade (which have originally been returned unclassified by the QVT parser) at the beginning of the computation. This helps to separate the concerns by removing such functionality from executors or entry class (main class). The categorization is done using three lists (according the described groups) that allow the polling of contents and removing of executed Relations. An additional group is used for storing all executed Relations for traceability reasons.

Serialization and Deserialization

Not only the creation of a Transformation Net is necessary, but to serialize/deserialize and manage the models representing the object information creation the course of execution of this application needs to be handled. Hence, several classes exist to bundle this functionality at a single point. Although the deserialization

and serialization is only done once in the whole execution, the bundling in an own category seems necessary, because their functionality does represent semantically different sorts of actions. The other areas are fully object-oriented and do only handle object representations of models – reading from or writing to them. The serialization process, however, provides the functionality of storing the objects to a real model file. This separation leads to good hiding of implementation complexity at some points of the application.

6.3. Execution Process

The actual execution process of Grade is split up in four major areas of interest. First, the state-intensive general mappings creation of TROPIC elements from QVT-R code is introduced in different granularity levels. Second, it is important how to integrate the information to accomplish the execution flow. Third, the model information (the state) of concrete models have to be added like it is done in QVT-R itself. Fourth, the actual usability of the Grade application is improved by adding a pretty printing functionality.

6.3.1. Course of Execution

The actual execution process is a complex stateful procedure moving from state to state by calling actions by action. As described before there is a need of reusing many code elements and, therefore, the execution process is definitely dynamic. This can be seen similar to rule-based approaches. The combination of used parameter values leads to a certain way of execution. However, changing one of the two parameters and calling the same code lines again (maybe through iteration or recursive approaches) often results in a very or even totally different result. So it is the aim of this section to introduce the course of action for an execution process in a medium granularity level to make the process behind understandable. Naturally, many essentials details have to be left out, to reduce it to an understandable issue.

Pre- and Postcondition

The actual graphical debugging process consists of three steps: The parsing of the QVT Code, the creation of the Transformation Net (implemented in this thesis) and, finally, the loading of the model information (and additional optional steps). All three steps are made available in the Grade application. However, for step one and three only the usage and integration was implemented in the Grade application.

First, the model for a certain input QVT-R code has to be created using the discussed QVT parser. This step returns an XML file (the model) that is then used as data basis in the following steps.

Second, the Transformation Net is produced. This process is complex and is discussed in all following sections representing the course of execution.

Third, After the Transformation Net has been built, the model information has to be loaded to the produced Net. This is undertaken using the TROPIC

plugin with the aim of integrating it as much as possible in the Grade application execution.

Execution Overview

The activity diagram from Figure 6.2 shows the basic iteration process on high granular level for the Transformation Net generation. This means that it concentrates on the selection of Relations and does not jump into depth of execution. The names of activities start with a letter, a number and a colon are diagram “ids” to simply the process of referring to a certain activity. This handling is continued in all other used diagrams as well.

The execution starts by classifying the Relations in groups as discussed in this thesis. Then the Relation groups are iterated (in a foreach-loop manner). Relation groups are the classification groups – top Relations, top Relations with When-Clauses and all other Relations which were not executable at there last call (pre-conditions not fulfilled) – and are stored in an ordered form. For each group the Relations are retrieved and iterated as well. For all executed cases these three classification groups have to exist, but they might be empty. At the start of execution the group of all Relations that are not top level is empty. This group is filled in the process of the execution. Relations that are only addressed over Where-Clauses are not listed in any of these three groups at the beginning. Only Relations that could not be executed at their call are added to the third group. This classification can be rather seen as entry point storage, because it is not important which Relations are existing, but it is rather important which have to be called to start an execution process – the root of a tree or subtree. For this reason, the third group need not contain information, if no post-conditions are used in all transformation mappings.

In the following “General process” the execution path within a Relation is prepared. Hence, it is in the full charge of the transformation process. The main Transformation Net generation is not done on this level, but rather on the level of Domains and properties. However, the registration of Where-Clause types is done and the delegation of the execution process is provided. This means that all related Relations (post-conditions) have to be called after the execution of this Relation as well. This is done according to the structural dependencies of this treelike graph. Moreover, the basic setup for execution is needed. A Module that holds the transformation visualization elements of this Relation and a main Transition has to be prepared and registered. The storing of data is done in an execution card and the related container objects. The setup of this storage environment is not done at this point, but is completed here for this execution stage.

So the Domains of the picked Relation are retrieved. Then another iteration process is used to execute each Domain. In this process the first task is to retrieve the DomainPattern that actually provides the relevant information of this Domain - all data needed to execute the process. In the following steps the execution of Domains and related items – or to be more precise their Patterns and values – are accomplished. Then it has to be checked, if the complex type related to the specific Domain has been passed over a Where-Clause to the actual Relation. This differentiation has to be down, because the Transformation Net production

6. Realization

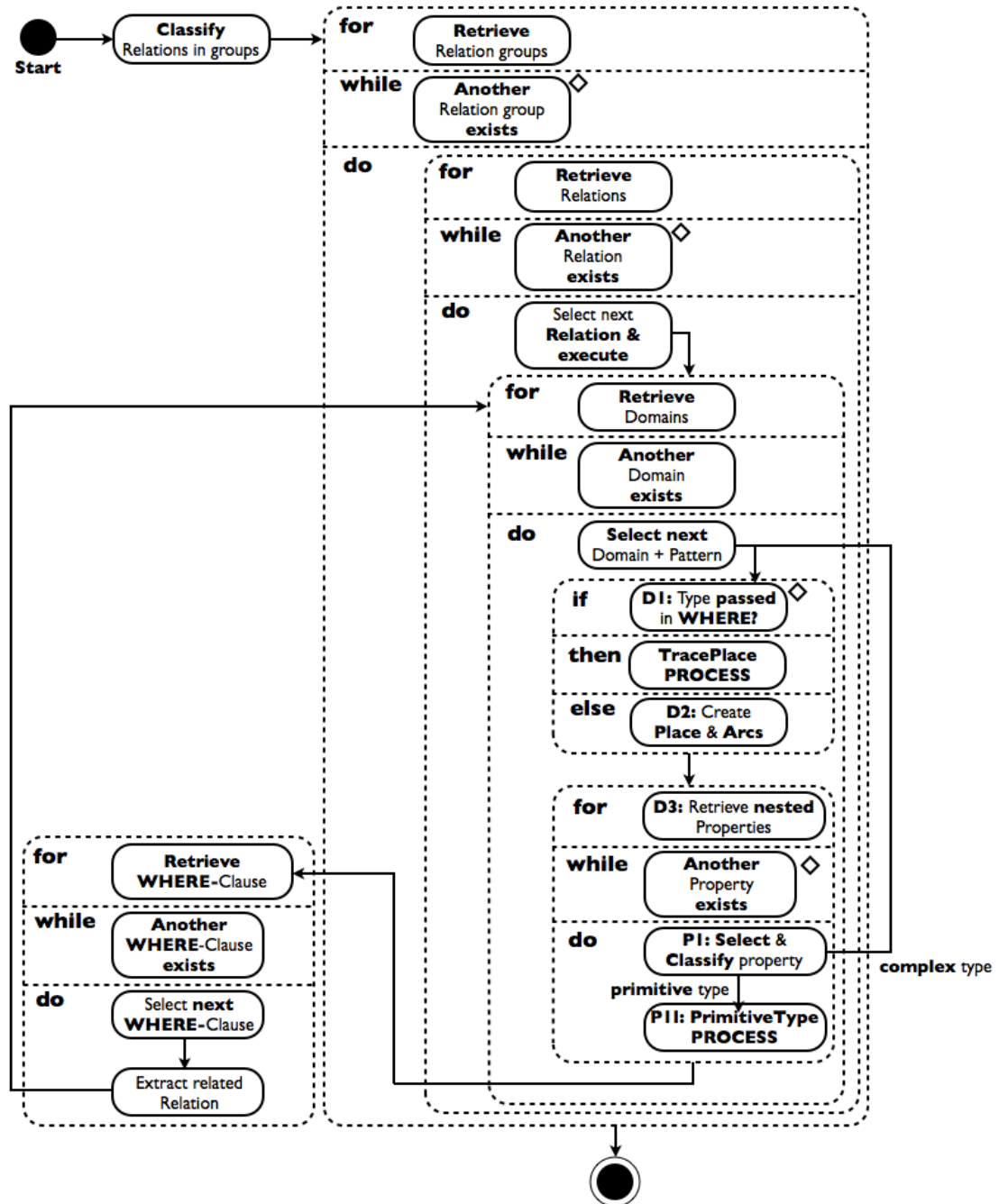


Figure 6.2.: High granular course of application execution

is different for these types. A type that has been passed in a Where-Clause (D1 is “YES”) and should be executed now for this Domain, involves a TracePlace and should avoid the production of the normal TROPIC elements. Therefore, the incoming information from the TracePlace is only a condition. The information of this incoming colour is reused to link related items to it. On the other side (D1 is “NO”) the Places (D2) have to be created newly. The Places have to be linked with the Transition of this Relation. At this point it seems necessary to mention that the TracePlace creation should only be done once for source and target side. Because source side is normally the first side to be executed, it is restricted to the source side.

The execution continues for this Domain by retrieving the nested properties of this complex type – the child elements in the perspective of XML representations. These child elements can have a count of zero or be infinite. Hence, it has to be iterated over all properties again. For each identified property another classification has to be made. At this point it is relevant, if a complex or a primitive type is found in the pattern of the property. Nevertheless, in the execution process on a more detailed granularity level it can be seen that several more constraints have to be considered to execute these properties. However, to represent the basic dependencies and flows the differentiation between complex and primitive types seem to be sufficient. The latter one can be directly executed to TROPIC Places, MetaTokens and Arcs. The first one points back to the point of D1, because the execution process is similar to Domains and can, therefore, include many more properties and execution iterations.

After executing every property of a Domain - and all the nested properties of a property – the execution of the Domain finishes and the next Domain starts to be executed. After transforming all available Domains the next referred Relation (Where-Clauses) can be executed – this is not specified additionally in the figure. When also all referred Relations are completely executed the next Relation can be used. When all Relations of a group have been executed, the next Relation group can be taken. After executing all groups from highest to lowest priority, the transformation finishes and the Net can be serialized.

For the process of developing the graphical debugger it is relevant that a conceptual or algorithmic mistake is often not reflected by exceptions during executions, but at the process of serializing the Net. For example if an element refers to another item, but the later one has not been added to the Transformation Net at any point – it can be placed in other items or in the Transformation Net itself. Therefore, the development and the handling of error cases can be somehow misleading. This represents a important issue for the future, but cannot be linked to this implementation approach itself, but rather to the usage of Ecore-models serialized to XML files. This serialization process already implies that the stored object have to be persisted and validated at the end of the execution process. Although it would be wishful and worthy, a parallel validation for objects cannot be provided, because it would need some transactional logic to understand when a validation could be done. This is a drawback that has to be accepted.

Furthermore, it has not been mentioned at any point that preconditions (using When-Clauses) are used and handled. These preconditions, however, can strongly influence the course of execution, because they can avoid a Relation to

be picked. This might not only be at the point of selecting the next Relation in the iteration process, but also in the course of calling referenced Relations from a Where-Clause. At all points of taking and executing a Relation the When-Clauses have to be checked. There is no need to prove, if this condition has been fulfilled with a particular set of data, because the graphical dependency in visualization is independent from the actual transformed object – the TracePlaces provide this process of delivering the correct data to the related Transition. If a precondition is not fulfilled at the execution time, the transformation of this Relation is skipped and added to the last Relation group. The last Relation group is picked, because the probability of executability rises by letting pass some other Relations. Often the circumstance occurs that a When-Clause cannot be fulfilled, because a Relation of another execution path (other top level Relation with or without preconditions) is required. As it has already been mentioned Relations only addressed by Where-Clause calls are not listed in any group at the beginning of their execution. However, if such a Relation has to be stopped in its execution and has to be skipped for later execution, the status has to be maintained as well. So they are stored in the third group like all others in the same situation - other Relations are moved from their original group to the third one.

This perspective on the process of this graphical debugger is comfortable, because leaves out the main execution. However, behind each step several other finer granular steps are placed and often more cases have to be considered. In addition, the noted structure does not fully reflect the structure of the application. It is the basic approach to encapsulate the execution process of each step, but it is sometimes more useful to reuse elements, especially those which are related to the production of the Transformation Net model. For example the used factories change the structural appearance of the application to a certain extent, however, they cannot be listed in the graph representing the execution flow, because this is too much linked to specific implementation issues. Classes used to produce the described flow, are often named or structured differently to handle the complexity on real basis more efficiently. This for example relates to the ideas of using “executor”-classes and similar approaches.

Detailed Transformation Net Creation

This level of granularity gives an idea of how the programmatic execution can be pursued. A more detailed introduced of the done computation is presented in the activity diagram of Figure 6.3.

The process starts by receiving the Domain. From this Domain the complex type – the related pattern – is extracted and used for the further execution. The next step is to look up the hierarchy level object. This can only be done by knowing the actual hierarchy level index. This information has to be prepared before calling this sequence of activities. This is also needed for the main Transition and the creation of relevant Modules.

Then it has to be checked, if the current hierarchy level is higher than zero or is zero. If it is zero (`higherLevel?` is “False”) than it is an execution on the same Transition and hierarchy as the beginning complex type – this should always be a DomainPattern. Otherwise it is the execution of a one-to-many related property and, therefore, holds an own Transition. This case needs further treatment,

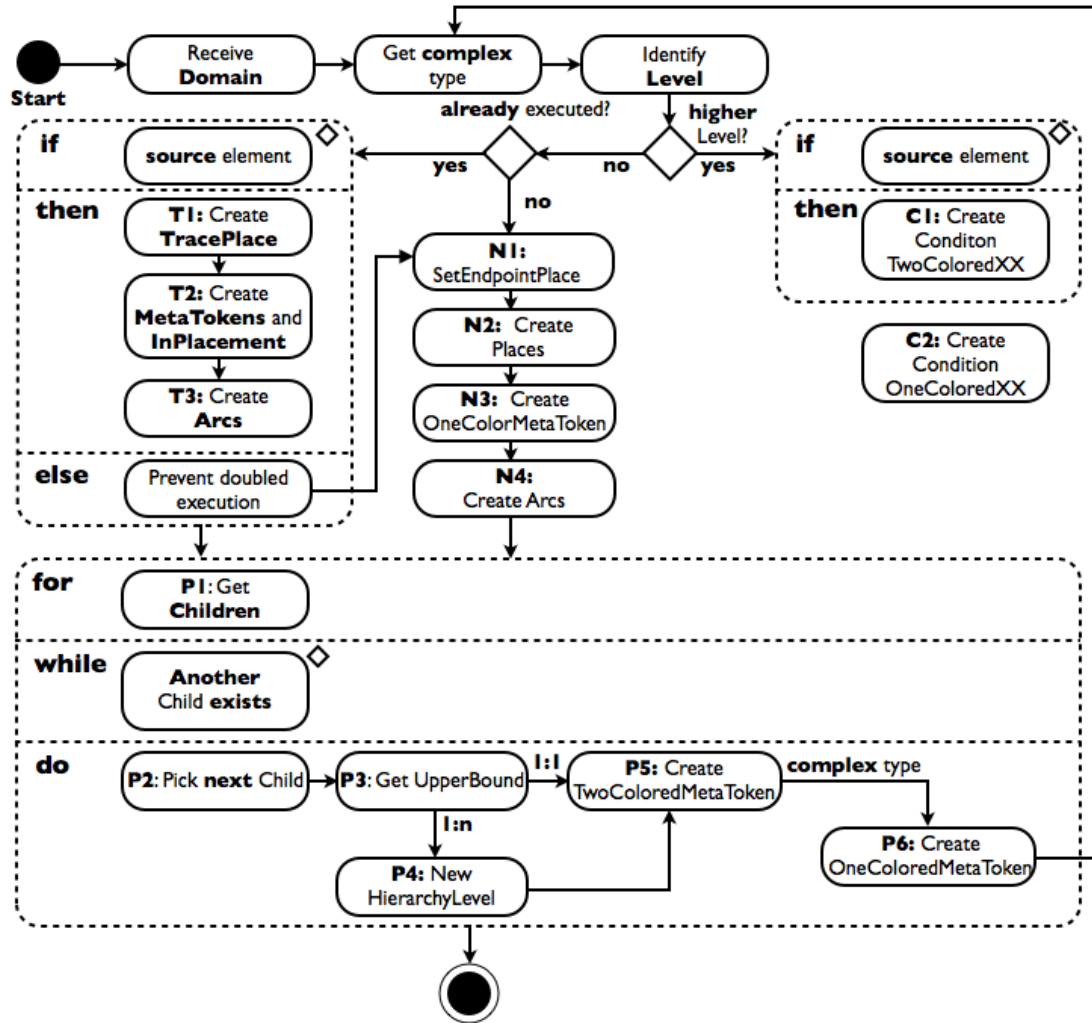


Figure 6.3.: Finer granular course of Domain and property execution

6. Realization

because both Transitions do not only have to be linked to each other, the parent Transition also needs conditions using MetaTokens preventing an execution of too many Tokens - too many Tokens typically refer (like in this situation) to too weak conditions. So it has to be checked, if it is a source or target side execution, because such a conditioning should only be done once and, furthermore, be placed on the source side. Therefore, for all source Domains fulfilling higher hierarchyLevels a OneColorMetaToken and a TwoColorMetaToken are produced in the activities of C1 and C2. The colouring has to be taken from the hierarchy level of both Transitions. The masterMetaColor of the parent Relation can be used as masterMetaLinkingColor – as described in the conceptual approach – of the child level (the detailed colour information processing is done like described in the concept and is only mentioned in the realization to give an idea where and when this has to take place). All executions of higher hierarchy levels continue with the execution of the basic complex type transformation done in the activities starting with N. But before describing this execution other paths leading to these activities have to be discussed. All target side complex types on hierarchy level zero also join the computation of this group of activities. However, this need not be the case for source side complex types on the same level, because they could be relevant for receiving information from TracePlaces. Hence, it has to be checked, if this type has already been executed and has been passed using Where- or When-Conditions. If this is not valid, the execution again joins the series of N-activities. Otherwise the T-sequence for TracePlace execution is processed.

The N marked sequence of activities cares about the basic complex type execution. This means that the EndpointModule (Source/Target) has to be retrieved and stored. Furthermore, they create the OneColoredPlace in the EndpointModule, if it does not already exist. If it is already available, it can be reused. Then the OneColorMetaToken is produced in the Transition of the actual level – with the additional In- or OutPlacement. In the last step of this series the Place is linked with the MetaToken using an Arc.

The T marked series reuses information passed from one Relation to another. This does not handle the information somehow passed within a Transition. For this reason these activities look up the related Transition and create a TracePlace (if not available for this Relation-to-Relation combination). Then again the MetaToken and InPlacement has to be created. The information is then combined by creating two Arcs - one from the Transition of the parent Relation to the TracePlace and the other one from the Place to the new Transition (The Transition of the parent Relation means looking up the Transition of the hierarchy level of the parent Transition with the highest level count).

Both execution series join and continue with the execution of child elements (P1). Naturally, not only one property can be included as child element, but rather a set of items. Therefore, the next child element has to be picked for executions. The ordering, however, is irrelevant, because both are executed on the same level. If one execution should need the usage of another hierarchy level, this does not affect the other properties, because they are all provided with the same hierarchy level information.

For the selected child element the pattern has to be received (the actual value). This pattern always has the type `OcIExpressionImpl` and can, therefore, be

different in the actual execution process. The most important step is to retrieve the upper bounds of the element multiplicities (`upperBound`) of the child itself (and not from the property) – the `upperBound` and the name are the only relevant information used from the property objects. One-to-many related properties can directly continue their execution with P5.

One-to-many related properties, however, need a higher hierarchy level, because they do not need to handle one object, but can need to handle a magnitude (up to infinity!) of objects that have to be treated fully independent from each other, but have to hold the relationships to parent and child elements. Because every one-to-many related property needs the execution information (in the visualization only the colour of the `MetaToken` is important) of the parent `Relation`, a `TracePlace` has to exist between these two `Transitions`. Moreover, the own `Transition` provides the independent execution environment. In P4 this new hierarchy level is produced, as well as the referred `TracePlace`, the involved `MetaTokens`, `In-` and `OutPlacements` and `Arcs` from and to the `TracePlace`. Therefore, this step is in reality a powerful and important one that needs to handle the colour information of both sides carefully.

Both sides join again in the activity P5, but one-to-many related properties already execute this information on their new level – the others stay at the same execution level as before. In this activity the `TwoColorMetaToken` is produced for the property to combine the parent type – whether on the same or different `Transition` - with the new `Type` by using correct colouring. The border uses the colour of the parent type (`FromColor`) and the inner circle color (`ToColor`) is the colour of the newly translated property. The `MetaToken` is placed in an `In-` or `OutPlacement` of the `Transition` of the current level. Furthermore, a `TwoColoredPlace` has to be created – if it does not already exist – in the `EndpointModule` and needs to be combined by using an `Arc` with the `InPlacement` of the `Transition`.

If the property is of primitive type, the property execution is finished at this points. Then the next property can be executed as the previous one. For complex types the execution continues with two further steps – one of those can be a really huge one. First, a `OneColoredPlace` (`EndpointModule`) and a `OneColorMetaToken` (current `Transition`) have to be created and linked – if they are not already existing. The colour information is used from the `ToColor` of the previously created `TwoColorMetaToken`. Then the execution continues with the execution of this complex type. This actually is the same execution process that has been described in the whole procedure above. At this point it is notable that the new hierarchy level status has to be passed on to the execution process. In addition, it can now be the situation described in C1 and C2 that add the condition to the previous `Transition`.

6.3.2. Managing Flow Information

For being able to fulfill the described flow of execution, it is necessary to retrieve certain pieces of information. As already presented in this thesis the differentiation between different types is possible checking the type of a property instance. However, it has not been discussed so far, how to retrieve the correct names,

6. Realization

boundaries and types based on the certain meta model.

The names are well integrated in the parser data. They can be received from any object, but the results can be more or less useful for a certain computation. To retrieve the name of a property, returns a string of characters like `classes` representing the name of the attribute within the model. This does not reflect the name of the type of the attribute or the name of a related variable. Hence, it is necessary to retrieve the pattern information – the value – form a certain property object. This object includes the type information and the related names.

For the execution of elements linking a parent and a child type, it is necessary to receive and store the name of the parent type (e.g., `Package`) and the name of the type stored in the pattern of the attribute (e.g., `classes`). Both names are necessary to uniquely couple the elements together. Uniquely in this context can be somehow misleading, because a second constellation with the same naming could occur. In reality this synthesis of names is sufficient for avoiding duplicates. The usage of a simple name of the type of the child property pattern cannot be useful, because the naming of properties is often similar in different types. For example `name` could be a property of nearly every type. However, `Class_name` is clearly understandable and well restricted.

Besides the correct retrieval of names and types, the correct upper bound is important for visualizing the model transformation. Typically for models there exist several different upper and lower bounds. For example, the pattern of a property offers such values, but – this is a common mistake – they always have the value of one. This results from the constellation that every property can have a boundary from one to infinite. The values of the certain property instance (pattern) is related to just one instance and represents the full set of data stored for this item. Therefore, the boundary can never change and is not valuable for the differentiation in one-to-one and one-to-many relationships. The boundary of the property itself – the `OclExpression` - represents the expected information.

6.3.3. Managing colours in TROPIC

The previously discussed `ColorManager` is a class reused from the TROPIC plugin that is able to create unique colours in the format needed to produce this Petri Net. To be able to manage the colours more effectively, methods were created in the hierarchy level that take the `masteringColor` or `masteringLinkingColor`, if they exist. Instead of returning null values a new colour is requested and stored as the certain item. Therefore, there is no need to explicitly think about the availability of a value for each hierarchy level. It can be used as existing, because every non-existing value obviously has to be produced as it is requested at this point. Therefore, this can be seen as a form of lazy-colour-management.

6.3.4. Adding Model Information

By now, the development of Grade focused on developing the correct Transformation Net. Indeed, all the done steps are necessary, but after running the debugger for a certain QVT-R Code only a Transformation Net without any Tokens results from it. This is the case, because at no point the model information of a certain

source model has been used until now. Hence, this information is not known for the execution. Naturally, the Transformation Net itself has to be independent from the data filling it.

To be able to use this debugger for real world development it is necessary to fill it with the same data the QVT engine receives to undertake a certain transformation. By now, the transformation direction is specified, but to be able to add the information by placing `OneColoredTokens` and `TwoColoredTokens`, the model information and the used metamodel are needed.

The TROPIC plugin is able to load such model information, but the plugin cannot know which Token should belong to which Place. Therefore, it can only be done by using a certain naming strategy that has to be used and implemented by the Grade application in the process of producing Places.

Naming Strategies and Conventions

For the production of a useful Net the naming strategy is important. The TROPIC elements do not use a unique identifier (ID) that differentiates them from each other, but a string of characters that should reflect the meaning of it. It is mostly interesting for Places that hold some information. They should be named after the complex type that is related to it in the model's metamodel. Therefore, typically this naming strategy can only hold for `OneColoredPlaces` representing the complex types and not primitive types or linking Places. Hence, this has to be enhanced to represent primitive properties and linkings. The name of two-coloured Places resulting from them should consist of the name of the parent complex type, the infix “_” and the name of the property itself – not the type name of the property. This leads to names that look like “Package.classifiers” for a complex type “Package” holding an attribute with the name “classifiers” that is of type “Class”. So it is not important which type name the property has for the production of the `TwoColoredPlace`, but it is needed to create its own `OneColoredPlace`, if it is a complex type itself. Moreover, it is not important if a one-to-one or one-to-many relationship exists, because the naming of Places only reflects the name of the containers holding the information.

`TracePlaces`, however, do not follow this naming strategy, because it is not dependent on any external information loading. It receives the information directly from a Transition. This Transition consumes the data from some Places that have to be filled. Because the data in the `TracePlace` is no new information, it need not be filled externally and, therefore, can be named independently.

The naming of Transitions and Modules are independent from this strategy as well. However, they should follow some rules to allow the developer a fast recognition of relevant pieces of information. For this purpose the Modules are always named after the Relation they belong to. The contained Transitions should always reflect the name of the main type of it.

Another useful naming convention has been defined for Arcs. They are always named according to the “work” they are doing. If they link a Transition to a `TracePlace`, then the name should include “ToTracePlace” and “FromTracePlace” for the other direction. Similar naming strategies can be found for every Arc usage, but they do not help the developer in using the Transformation Net for debugging. However, they can help in the development of this graphical debugger,

6. Realization

because it helps to realize from where the information should come and to which Place it should flow. It also helps to find the correct Arc to test some constraints or qualities. These names are, however, not visible in the TROPIC view itself, but are very useful in the XML representation (which can be used for debugging or other tasks) as they make the differentiation easier. Therefore, it is a quality characteristic that should not be omitted.

Loading models

This is done using the TROPIC import functionality that is included in the tool palette of Eclipse after starting the plugin. Hence, the **LHS Metamodel** has to be specified, which means the left-hand-side metamodel. In the context of this thesis LHS is always called the source side. The metamodel is needed by the plugin to fulfill the naming needs discussed above.

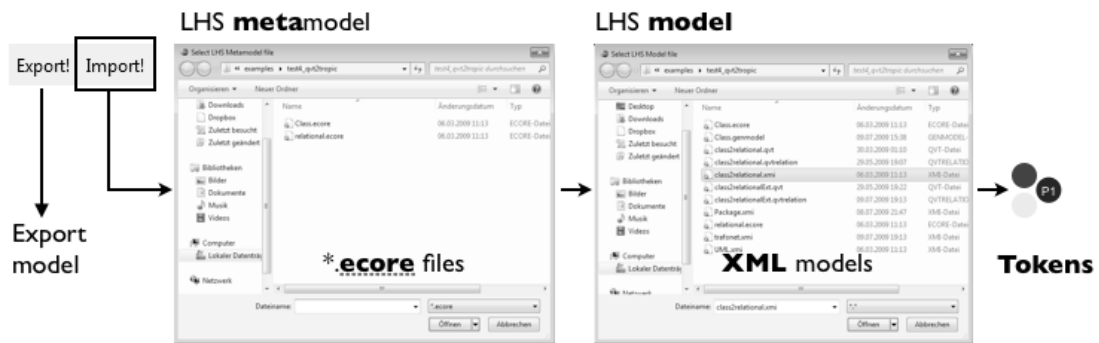


Figure 6.4.: Import and export functionality in TROPIC

Furthermore, the actual data should be loaded from the model. So the **LHS model** has to be specified. Then Tokens are produced according to the data included in this file. They are placed in the Places with the names using the discussed naming strategy. If one Place has a wrong name, the Tokens cannot not be produced and placed there, which actually means that the information reading is unsuccessful to some extent. Figure 6.4 demonstrates the model loading process in TROPIC by describing the dialog sequence after calling the functionality in the application. The export functionality of TROPIC is not used in the context of this thesis.

After reading the model information it can be started to test the Transformation Net. By firing all possible Transitions the information should flow from the source to the target side. Therefore, the actual model transformation is rebuilt and visualized. The real debugging process can start after doing this loading step, because it is then possible to visually follow the process of executing one mapping by each other. The information, therefore, moves from the left-hand-side to the right-hand-side using some Transitions and intermediary states. At all points of execution the developer can analyze easily from where the information came from, how it is related on other elements and where it flows to.

Without using the model loading process, only “dummy”-testing can be done by placing some Tokens for testing purpose on the Net that do not reflect any real information. Naturally, the real information stored in the Tokens can be rebuilt

by hand, but there can never be a trust that the developer made it correctly. The automatized process is fast and tested and, therefore, similar to what a developer could expect from a visual debugger.

Inheritance The model loader provided by the TROPIC language is not able to fulfill the adding of model information in the handshake process – structure creation and adding the actual state – of fulfilling types involving inheritance, because it does not handle inheritance at all. Hence, the used loader has to be extended.

For every object of a certain type colours are created and afterwards set to newly added Tokens. So in the process of retrieving the object and its type, all supertypes have to be retrieved. For every supertype the same colour as for the subtype is added. This is a unidirectional constellation, because instances of the supertype cannot be reflected to any subtype without any clear evidence that is definitely an instance of this type. This can only be proved by holding an object of the subtype. The adding of colours for a certain type of a model leads to the production of Tokens for every stored colour, if a Place exists. By adding the colours for a type, new colours are produced. So the duplicate values of super- and subtype are produced automatically.

Beside the modification of the loader, some errors were identified by the usage while developing the extensions. The colouring was erroneous at some points, because zero values were set. This led to the generation of another randomly created colour information. After duplicating it for the inheritance structure it was visible that the colours were unintentionally set, because the generation of a new colour led to a situation of different colours of sub- and supertype, although the same colour was set originally. The errors were removed and can, furthermore, express the importance of this practical implementation to the quality improvement of the TROPIC plugin itself.

6.3.5. Element Arrangement

The element arrangement and other pretty printing mechanisms are not in the focus of this thesis, but are handled to some extent to make the resulting Nets useable for further developing this prototype and to evaluate the provided TROPIC Nets.

There exists an element arrangement mechanism in the TROPIC framework that is in several ways insufficient for the usage in this thesis. Programmatically producing a Transformation Net with this mechanism results in a dozen of overlapping elements, because they have to be placed without any positioning information. The arrangement functionality provided by the TROPIC framework is buggy (NullPointerExceptions for missing names, unintuitive placement of elements, high distances between related items, etc.). The original variant did spread the elements widely over the whole canvas without any knowledge about dependencies and semantical or syntactical relationships. Furthermore, the EndpointModule logic used in the Grade application was naturally ignored. Although the errors could be removed and some quality improvements could be achieved, the mechanism has not worked adequately as it still could not arrange related

6. Realization

items close to each other.

So, a new layout provider is used in Grade that optimizes the element arrangement for the debugging usage of QVT-R and tries to overcome these pitfalls. Therefore, it positions source and target Modules on the flanks of the Transformation Net. In the middle of the visualization all other Relations in form of Modules are placed. However, it is useful to place only one Module (besides source and target) in one row (same y-coordinate). Source and target Modules can be considered as columns interacting with all other Modules in the center and, therefore, typically range over the whole Transformation Net canvas.

The size of each Module is automatically adopted by analyzing the contained elements like Transitions and Places. The more elements are used the bigger the Module has to be. This also influences the positioning of all following Modules. According to the placement mechanism used in this thesis the growth of the height of each Module is more important than its width growth.

Within one Module the arrangement should be optimized as well. Therefore, two modes are defined. On the one hand the “Source-Target”-mode that arranges all Places (or other elements) in one single column from top of the narrow Module to the bottom – placing only one element in each row. In the source and target Module typically only Places are needed, but other elements would be placed in the same manner. The second mode is responsible for all other Modules placed in the center of the Transformation Net. These broad Modules hold the Transitions and TracePlaces related to execution of the Relations and its Domains. The transformation execution has to be geared to the placement of Transitions. Therefore, these Transitions are set in the first column – at most one Transition should be placed by each row of a Module. In a second column Places are set. They typically directly refer to the Transition placed before and, therefore, seem to be appropriately positioned with a lateral-offset to the Transition for being able to interact with several ones. Moreover, it is important to set Places in a new row, if the column for Places has already been filled by another Place. However, the column (the x-coordinate) has to stay the same to allow easy recognition of structurally similar elements.

This new layout provider allows a quick overview of the TROPIC Net and arranges elements closely together. The distance between related elements could be dramatically reduced as it respects the domain specific usage of TROPIC.

6.4. Summary

In this section the implementation specific details were presented including the tooling needs, configurations, and modality and structuring of the implementation. A modern tool set including a modern Eclipse IDE version – with a set of plugins – is used that provides good extendability for the future. Furthermore, existing project such as TROPIC and a QVT Parser is used to avoid a “reinvention of the wheel”-methodology in the development of this application. Therefore, the focus could be led on the most important issues such as the mapping of the QVT-R structures to TROPIC, the presentation of results and the handling of simple incremental changes.

The execution provided by the Grade application is strongly state-dependent

and iterative. Therefore, the overall process is rather complex, especially resulting from the reuse of many logical blocks. This reuse cannot be avoided according to the deep and broad nesting that has to be supported in QVT-R. Often slightly different transformation variants exist such as the Where-Clause TracePlaces that differ from TracePlaces resulting from one-to-many relationships (in visualization and especially storage of information).

6. *Realization*

7. Evaluation

In this section a retrospective on the discussed approaches, used technologies, and implemented features will be done. The focus will be lead on evaluating the suitability of the presented approach and to identify open problems.

7.1. Capability of Erroneous Code Recognition

In this section general problems in the usage of QVT-R are identified and compared with the capability of Grade to help to identify them. First, some general considerations are presented and, deepened, with exemplary pitfall detection examples.

A typical error is to forget some necessary elements in the QVT-R transformation. This means that for example a certain condition is needed that filters a certain type. Identifying such a fault can be difficult without debugger, because the transformation execution path is not visible. Therefore, the developer cannot know at which point too many or too few elements are present. All information of the source and target model, as well as the transformation process, have to be kept in mind to be able to expect a certain output and to compare it to the produced result. The visualization in TROPIC can help to identify such issues as it can be analyzed that a wrong number of Tokens are present at a certain Place.

Another typical and hardly recognizable issue is the specification of transformation details at the wrong position. For example if the parent Relation that refines the transformation has weaker conditions than the child Relation. Then the skeleton is produced in the parent Relation, but the refinement cannot take place. A similar problem exists by forcing a wrong ordering of the Relations by wrongly using pre- or postconditions.

7.1.1. Pitfall Detection

According to [15] several typical pitfalls of QVT-R can easily be detected in TROPIC, which will be discussed in this section shortly.

A characteristic scenario is used to show typical problems that can occur. The basic example scenario can be seen in Figure 7.1 and will be used as basic reference to define typical pitfalls. The used examples should highlight the qualification of Grade for debugging.

A **Package** can hold several instances of the type **Class** that directly depends on the **Package**. In the class the attributes **isPersistent** and **name** exist. These **Packages** are transformed to **Schemas**, if they hold at least one instance of **Class** where **isPersistent** is true.

7. Evaluation

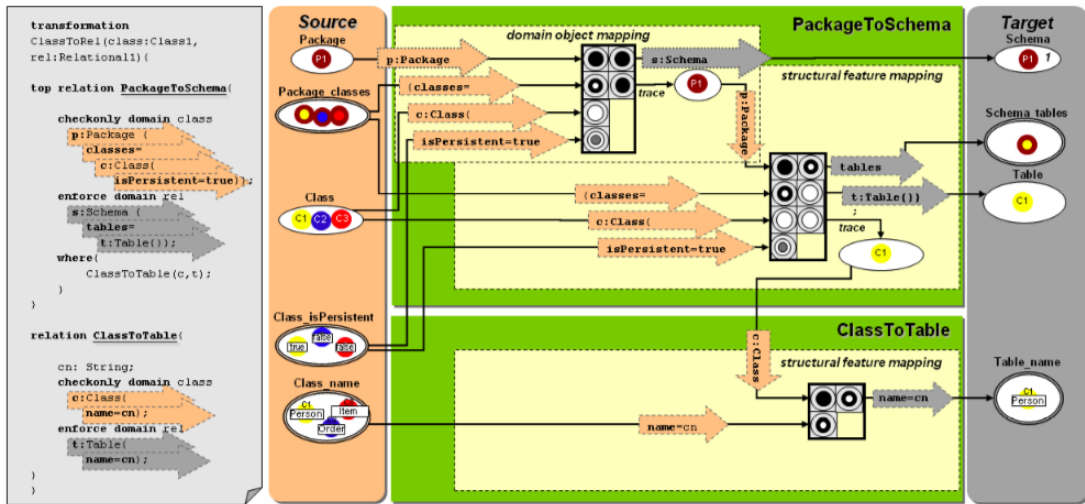


Figure 7.1.: One-to-one and one-to-many related properties transformation [15]

Interrelation Need Example

The first typical problem is the usage of several top-Relations that are not related to each other (referred Relation according When- and Where-Clauses). This results in completely independent data instances. Duplicate values can result from executions. For example a Relation is defined that executes Packages and its classes. The next Relation is used to express the detail of the classes. Therefore, the classes would be created two times for each instance that exists in the Source model. Detection: All instances of a type (here **Class**) occur several times in the target model Place. Furthermore, more than one Arc points to the target Place of the type **Class**. Solution: Top-Relations containing same data parts have to be related to work on the same instances (continued in the next Pitfall)

Adoption in Grade As it is described in figure 7.2 this pitfall is visible in the Grade application. The overall visualization is different in its pretty printing behaviour, but it is easily recognizable that two Arcs are pointing to a single target Place. If this is intended to be, the developer knows that this duplication works as is it has been planned. Otherwise this obviously a fault to be corrected. However, Grade cannot provide the same element distribution on the canvas like shown in Figure 7.1. Although the arrangement capabilities of TROPIC are enhanced by Grade, the result is different.

Condition Strength Example

After realizing the need of dependency and interrelation between two Relations (partially) working on the same set of data, the quality of interrelation has to be defined. This is done using an example of the discussed article [15]. The calling Relation can be too weak regarding its used conditions and in contrast to the called Relation. For example the calling Relation uses all instances of Type class whereas the called Relation defines further restrictions using conditions (for example **IsPersistent=true**). Detection: Too many instances can be found

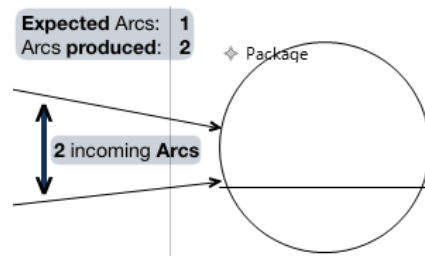


Figure 7.2.: Two independent producers of **Package** Tokens – one is intended and one unintended (Comments used as explanations have a light grey background)

in the TracePlace between called and calling Relation. Solution: Use the same restrictions/conditions for the shared type in the calling and called Relation or shift the condition from the called Relation to the calling Relation. This results in the same amount of writing, but less elements in the TracePlace. Hence, the condition `isPersistent=true` is found in the calling Relation. According to the referenced article this can be seen as correct solution.

Adoption in Grade This issues can be identified similarly as it has been planned theoretically. The TracePlace is filled by the user firing the Transitions. By firing the calling Transition it happens that items are produced that do not qualify for further transformation. Therefore, these Tokens will be placed in the TracePlace, but no execution takes place with them. Hence, referred objects will not be transformed. This is visible by identifying for which Token (colour) properties (FromColor – colour of the border) have been created. All other Tokens are result of the too weak condition in the calling Relation.

7.2. Visualization

The used TROPIC plugin is responsible for the graphical arrangement and automatic layout generation of the Transformation Net. However, in this application the encapsulation is handled to optimize it. Hence, source and target side Places and all relations specific transformations are all stored in special Modules.

However, this cannot change the way the Modules are placed on the surface. The TROPIC plugin does not allow the specification of coordinates, but it allows to use a variant of an automatized graphical arrangement method that is derived from the GMF functionality. This mechanism is not optimized for the TROPIC language and, therefore, leads to the problem that it tends to distribute all Tokens widely on the canvas to improve visibility, but does not care about the information flow. So, the useful placement of all items of the Net has to be rearranged by hand. This is enough for the an application in prototyping status, but would lead to enormous efforts in real world debugging scenarios. The advantage of the proposed graphical debugger is the time and quality benefit in handling erroneous QVT Relations statements. However, the time benefit gets lost, if developers have to concert their Net manually.

7. Evaluation

A second visualization problem is the scalability of these Nets. Declarative transformation mappings have a simple and straightforward appearance. The hidden operational semantics behind each declaration is considerably more sumptuous in details and instructions. The Transformation Net in the graphical debugger tries to visualize the operational semantics and, therefore, has to produce extensive flows of information. However, it can be seen as problematic that the growth of Nets cannot be considered to be linearly, but rather polynomial or even worse by adding another one-to-many relationship. See figure 7.3 for an example of an unsorted Transformation Net holding several Relations and one-to-many related properties in their Domains.

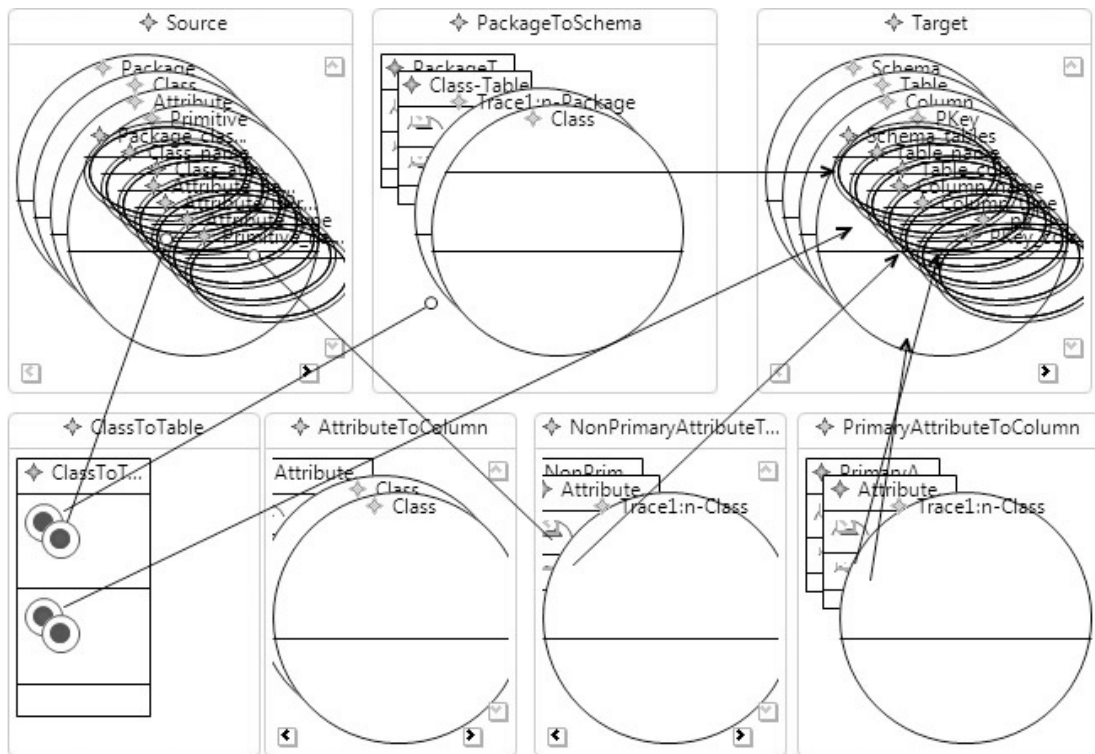


Figure 7.3.: Unsorted Net with several Relations and one-to-many-relationships

Nevertheless, the visualization of transformation mappings is clear and human-understandable. The transformation flow can be seen by following the path of the used Arcs from one input information to its produced and related outputs. Furthermore, it is easy to understand which type belongs to which Place in the Net, because the naming is well structured and uses the original specifications. The linking of attributes with parents is easily understandable as well, because the parent type name as basis is combined with the enclosed attribute name.

The approach of using colours has several advantages to using more sophisticated approaches. It helps to focus on the structural meaning and behaviour of the used constellation and keeps track of identities – each colour can be seen as unique identifier that is easily recognizable without looking in the details of a Token. Additionally, the focusing on the most relevant information for a special task is highly linked with the usage of models at all. Models give people an easily recognizable representation of real world problems by presenting a certain

useful perspective and leaving out (implementation) details. This is common way of reducing complexity to a human manageable amount. However, it has to be acknowledged that between four and five percent (calculated from several figures [34]) of all humans cannot recognize all or some colours. This fact is not addressed by the actual prototype, but can be done by using black-white patterns in lieu of colours.

For the discussed problem with the element arrangement – including its sizing and nesting – an own defined layout provider was created that arranges the items according to the specific usage in this graphical debugger – this can be interpreted as focusing on the most important arrangement issues in this context and is an alternative to the other existing visualization mechanisms. This layout provider has to address the problems of mixing source, target and intermediary Places with each other, TracePlaces not being placed between the two involved Transitions, and a generally unclear structuring.

On the flanks of the canvas, therefore, the source and target Modules are placed. In the center the Modules representing the Relations are positioned. This structuring dramatically increases the possibility of realizing the main problem areas and decreases the needed time to rearrange the Transformation Net by hand. The clear differentiation between Transitions and Places seems to be beneficial as well, because it is obvious where a developer has to look for to find a certain information.

However, in the used approach no analyzation is done according the correct placement of a TracePlace. A TracePlace typically relates to at least one Transition and should be placed closely to it. In the used mechanism this cannot be guaranteed. Additionally, the calculation of Module sizes is actually not done by including all facets from finest granular perspective upwards to the most coarse grained view. Hence, misinterpretations in form of too small or too big elements (width or height) could result from this fact. Moreover, it seems notable that optimal results could not be achieved – helping to realize all elements at once – for all situations tested in the development of this graphical debugger.

7.2.1. Performance

The performance of the graphical debugging process can be seen in several perspectives. On the one hand the time to produce a certain visual representation can be measured (parsing of inputs, TROPIC Net creation, serialization, synchronization, model loading). On the other hand the time to be able to recognize problems could be used – which is not deepened here as it relates to the common pitfall detection. Furthermore, it is necessary to acknowledge that the resource usage may be of high importance as well.

The time to create the TROPIC Net is good measurable and does only depend on the complexity of used QVT-R mappings. For examples that have been used and presented in this thesis the transformation process typical took less than one second with an additional serialization time of 0.3 seconds. This highlights that the performance of analyzing the transformation mappings is sufficient for the intended debugging usage. It also shows that the serialization of the model takes about 30 percent of the overall time. The stated time constraints, however do

not include the time resources necessary to open the resulting TROPIC Net and to rearrange it with the provided layout manager.

7.2.2. Unavailable Language Elements

The QVT specification is detailed, but the usable constellations are even more creative. Recursive information flows, passing of information at an unexpected position, passing of massive amount of data or other issues could lead to situations which produce Transformation Nets that do not seem to be optimized for this certain case.

The most recent problem is the need of interpreting the semantics of QVT-R language constructs such as `check` and `enforce`. The QVT-R specification does not provide such definitions. Furthermore, a certain interpretation is necessary in the identification of correspondences. This interpretation is a gap resulting from the restriction of being only able to analyze syntactical correspondences. In QVT-R, especially, complex type correspondences are often not specified in the code.

Additionally, QVT-R supports powerful OCL expressions. These expressions are often used in QVT-R to select certain elements of a set of elements or to transform it to very specific needs. According to the definition of QVT-R, all mappings can be considered as OCL Expressions. However, the discussed expressions in this thesis so far, have not fathomed the high potential of complex OCL queries. Therefore, it has to be mentioned that the visualization of such expressive queries is necessary for future research as well. Due to its powerfulness and its ability for long queries, a certain complexity results. The results of the used OCL expressions, furthermore, directly influence the used objects in the TROPIC Transformation Net. Therefore, the visualization of these expressions is important for understanding why a certain result could or could not be achieved.

Another unsupported feature of QVT-R is the usage of bidirectional transformations in one single TROPIC Net. The execution direction can be flipped such as for QVT-R engines, but the visualization is only done according one specific execution direction.

7.2.3. Effort Increase Analysis for Transformation Net Creations

To identify particular effort increases in the Transformation Net creation several common instructions are evaluated. Instructions are simple statements in the QVT-R code that mostly have the length of one line. Lines of code were not used, because often the formatting with curly braces that often cause the usage of several lines for a single instruction. An instruction typically finishes with a semicolon. For complex types holding several properties this has to be mitigated. All elements within the curly braces do not depend to the basic complex type instructions, but are own instructions – if there exist any. Naturally, the nesting using braces involves dependencies that have to be transformed, but this does not influence the instruction boundaries themselves.

| Identifier | Description | Items |
|------------|---|-------|
| Rt | Adding a new top level Relation (without consideration of reuses) | 2 |
| Rw | Adding a new top/non-top level Relation with called by a Where-Clause or possessing a When-Clause | 9 |
| OP | Adding a new one-to-one related primitive property | 6 |
| OC | Adding a new one-to-one related complex property | 12 |
| Ns | New complex one-to-many relationship | 26 |
| NnMs | Adding a new complex one-to-many relationship where parent Transition is not the main Transition | 32 |
| NAP | Adding a new primitive property in an 1:n related complex type | 5 |
| NAPp | Adding a new primitive property in an 1:n related complex type where the parent Transition is not the main Transition | 8 |
| NAC | Adding a new complex property in an 1:n related complex type | 10 |
| NACp | Adding a new complex property in an 1:n related complex type where the parent Transition is not the main Transition | 16 |

Table 7.1.: Description of the used effort increase scenarios

See Table 7.1 to identify the used scenarios. The perspective of adding a certain new instruction is chosen to identify the problem areas in visualization. Hence, these instructions can be considered as simple and plain that do not contain any further information unless it is stated. The stated elements in the table are the TROPIC elements that have to be created newly to add this certain feature to the Transformation Net. Such elements are Modules, Places, Transitions, Arcs, In-/OutPlacements, OneColorMetaTokens, TwoColorMetaToken. The dynamic data involved by filling the Transformation Net with concrete model information does not matter for the evaluation of the structuring. However, these considerations do not give the different TROPIC elements a weight according their complexity and size. The comparison of the discussed scenarios of Table 7.1 is shown in Figure 7.4

The most typical scenario is the adding of a new Relation in the QVT Code. Starting with an empty file it is necessary to add one or more Relations that contain the transformation mappings. A Relation called by a Where-Clause or a top level Relation with a When-Clause involves more transformation steps and, therefore, requires more new elements than top Relations without precondition.

In contrast to adding new primitive or complex properties that are moderately costly, one-to-many relationships involve many new items. The highest count of new items can be seen for the scenario of adding a new 1:n related complex type where the parent Transition holding this type is not the main (first) Transition of this Relation (Module). Such single instructions involve up to 32 new TROPIC items. If the parent Transition is the main Transition, still 26 elements have to be created. This implies that by adding a single instruction of such types the Net grows exorbitantly. Using many nested one-to-many relationships, therefore, will lead to a Transformation Net that cannot be viewed by humans anymore.

7. Evaluation

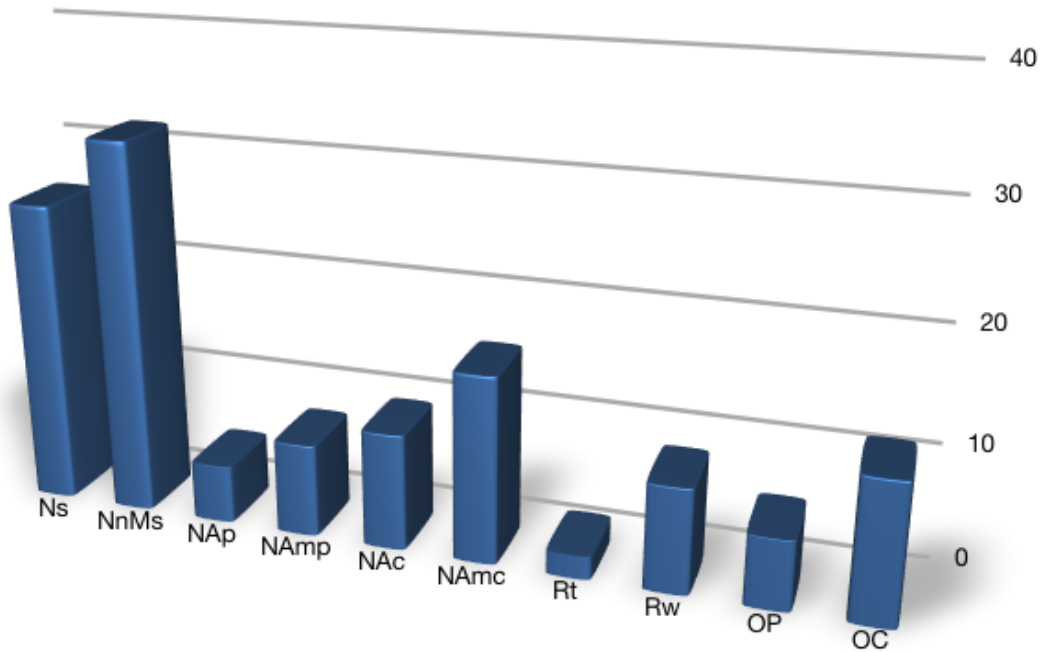


Figure 7.4.: A diagram representing the element increase for typical scenarios -
Abscissa: Different scenarios - Ordinate: Element count

Another interesting issue is the scenario of adding additional properties to such one-to-many related types. Here the increase is dominated by the facts, if the new property is a complex or primitive type and if the parent Transition is the main Transition of this Relation. The basic skeleton of complex types costs about 60% more than their primitive type equivalents. If the parent Transition is not the main Transition, the item count even doubles.

Furthermore, the needed space to visualize a certain Transformation Net item can lead to other issues. One-to-many relationships need a new Transition. Transitions are size-intensive and, therefore, matter most. Modules are needed for the creation of a Relation. Although they typically represent the biggest item on the TROPIC Net that can hold dozens of other elements, their usage is rather rare in a Transformation Net. Typically a handful different Relations are used and these Modules help to differentiate between elements depending to different Relations. So the main issue is the reduction of Transitions and MetaTokens/InPlacements. The latter one are interesting, because of their great numerousness.

7.3. Model Synchronization

In this thesis the advanced topic of model synchronization has been presented. Model synchronization is important for the ease of use of Grade. In this thesis only a simple solution synchronizing Tokens has been implemented, but several other use case have been presented. The simple approach is a good addition to Grade, but cannot be regarded as sufficient for professional usage. All presented

use cases in section 6 are desirable for professional debugging. This is an open issue that has to be solved in complementary work.

Nevertheless, it needs to be acknowledged that the identification of TROPIC elements by name is a major drawback for synchronization approaches and their quality. This should be modified in future version of the TROPIC language to be able to achieve high quality results with efficient solutions.

7.4. Summary

The main model transformation process works with good performance and produces a human-understandable Transformation Net. However, there exist several unsupported language elements, element combinations or derivations from the original QVT specification such as primitive Domains. Furthermore, the QVT-R standard does not provide a detailed inference of usages and their backend effect (undertaken by some engines) which forces the used approaches such as others to use some interpretations. The detection of typical pitfalls in QVT-R using the Grade application turned out to be as good as theoretically discussed. The most common mistakes are easily recognizable according to missing Arcs, duplication of values or missing values. Furthermore, the propagation of solutions for advanced and related features such as synchronization highlight the extendability of the concept, but undoubtedly highlight as well the need of complementary approaches. The advanced problems of class inheritance and synchronization presenting simple mechanisms. However, especially the latter problem again highlights the need of the overall application for a more compact visualization form.

7. *Evaluation*

8. Related Work

The debugging, testing and verification mechanisms for QVT-R code are rare. This section highlights the most effective error handling alternatives.

In particular related work from (1) the observation of erroneous code over (2) the tracking of origins of errors to (3) the fixing of bugs is outlined (cf. [32]). Verification concepts are finally presented to address unsolved problem areas of (1) and (2). It concentrates on formally proving or analyzing the code whereas testing concentrates on using certain inputs and expected outputs to handle the most typical usages. The focus in this thesis is on graphical debugging approaches. Some tools for debugging are presented as well.

8.1. Testing

Testing of code such as QVT-R implies the specification of test scenarios to be used for observing the code. Hence, it can be used to identify erroneous results (1) and often even to track the origins (2) of such errors. It is typically not the aim of testing approaches to provide mechanisms to fix bugs as this is rather intended to be realized by debugging in following tasks. Testing and debugging, however, cannot be seen as alternative, but rather as supplementary strategies.

8.1.1. Trial & Error

A human-like approach for error handling is the rudimentary trial and error approach (typically manual) as it is described in the work of Jochen Müsseler [18]. This can be applied for error detection in QVT Relations code as well. Trying to transform a certain model into a target model, a specific outcome of this process will be received. This outcome can be compared with an expected or desired result to detect errors. So, typically the code is modified after receiving undesired outcomes. Adaptions have to be decided manually and their effectivity is only tested by running the test case(s) again. After reaching the desired outcome the code seems to work correctly.

On the one hand the advantage of this approach is the simpleness. As it typically only involves manual invocations of the code, no additional tooling support is typically necessary. Furthermore, it represents an intuitive approach for solving problems. On the other hand the main disadvantage is the low efficiency, but also the low effectivity. Due to this time consuming approach of trying out code modifications as long an incorrect behaviour is achieved the efficiency is quite low. Effectivity is related to the problem that only “desired” mistakes are corrected. If there is a mistake in the transformations code not related to certain test data items, no mistake can be identified in the transformation result. However, it is desired that a graphical debugger (or a debugger at all) allows a perspective on

the transformations that enables to identify possible problem areas and issues not related to any tested data.

8.1.2. Test-Driven-Development

Pau Giner et al. [11] introduce a concept of Test-Driven-Development (TDD) for model transformations. TDD uses tests as entry points for developing applications. Ideally the checking of test cases works automatically. At the moment of writing no tool support for TDD for QVT-R exists.

In the discussed paper [11] Pau Giner et al. use the Human Usable Textual Notation (HUTN) [20] – which was standardized by OMG – to meaningfully express test cases. Furthermore, it is stated that HUTN is generic and could be “applied to any MOF-based metamodel” and would be fully automated and “no human intervention” would be necessary. As it is named “Human Usable”, HUTN is designed to be human-understandable and easy modifiable.

| Num | Version | Intent | Mapping: UML2DB |
|---|---------|--|-----------------|
| 1 | 1 | A concrete class generates a table with the same name. | |
| Test data | | Expected result | |
| Class "Class1"{ isAbstract: false } | | Inclusion: Table "Class1"{ columns: Column "PK_Class1"{ } } Assertion: "Has primary key" self.getPrimaryKey().isDefined() Assertion: "PK is the key column" self.getPrimaryKey().members.first() = self.columns->any(c c.name="PK_Class1") | |

Figure 8.1.: A HUTN example representation [11]

In Figure 8.1 there exists a class `Class1` in the section `Test Data`. This information is provided as input for the test case. On the right side there is placed the `Expected result`. It is mainly dividable in “result parts and assertions” [11]. In the result part the expected result can be defined to be “inclusion, exclusion or exact” [11]. The assertions are named OCL-Queries that have to return true. If the expected result for a specified input matches the outcome and all assertions return true, the test case is correct.

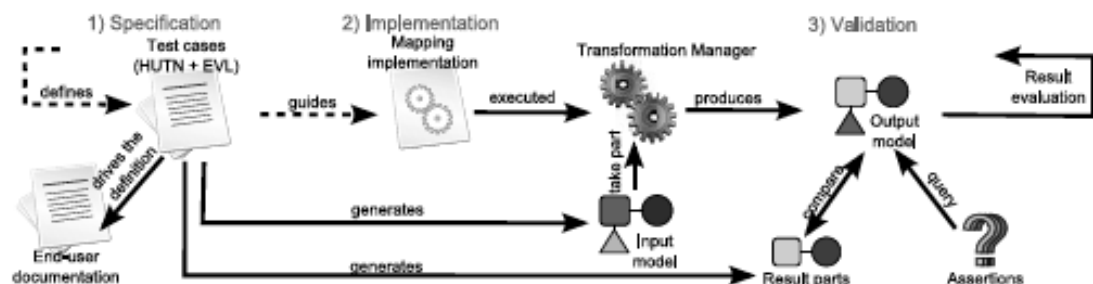


Figure 8.2.: The development cycle using TDD for model transformations [11]

The basic procedure of TDD in the field of model transformation is sketched in the Figure 8.2. Starting by defining test cases in HUTN the input models and result parts for validation are generated. Then in the second step the implementation of the model transformations code takes place. The produced source code and the generated input models can then be used to produce a certain outcome. This result is compared with the expectations from the generated result parts. Moreover, the assertions are run at this stage. This leads to an evaluation of the source code and typically results in further source code development or improvement.

8.2. Debugging

Debugging is a very commonly used method for tracking the origins of errors (2) and, furthermore, fixing bugs (3). A technical tool, application or sometimes plugin for an application providing debugging functionalities is called debugger. However, the terminology of a “graphical debugger” (or “graphical debugging”) can be somehow misleading. The debugger offers a graphical representation of variables and data structures necessary to debug a piece of code. Typically this involves a variable inspector as it is used for many programming languages like Java or C++ in IDEs (Integrated Development Environment) like Eclipse. The term “graphical” results from a graphical user interface (GUI). As it is aimed in this thesis the same terminology refers to the graphical visualization of the data involved in the MT. So the visual debugger is seen as a mechanism representing causal information visualizations of why a certain result is achieved or a certain error occurs. However, as it should be the intention of a graphical debugger to support the identification of problem areas and their origins, this section focuses on this aspect.

An example for a classical graphical debugger is the DataDisplayDebugger¹ (DDD) which allows the inspection of program variables in intermediary execution states by placing so called breakpoints. These breakpoints stop the application execution until the developer manually allows the execution to continue. Meanwhile, the developer can inspect the actual data status. By setting many breakpoints the developer can get an overview of how the data changes and even in which line of code this is reflected. However, this does not automatically allow a causal inference to the problem origin. Furthermore, DDD is known for the “interactive graphical data display, where data structures are displayed as graphs”². The most known debugger is the gdb (a GNU project debugger)³ project which is the basis for many other top-level debuggers.

Another interesting characteristic of debuggers is the ease of live or forensic debugging. Live debugging means the direct and immediate interaction of a certain result and its source code (and potentially inputs). If a certain result – failure or success – is returned the code can be modified for another test run. Forensic approaches, however, analyze pieces of information that are produced in the action of executing the certain code. Typically such pieces are log files and

¹<http://www.gnu.org/software/ddd/> - last accessed: November 27 2009

²<http://www.gnu.org/software/ddd/> - last accessed: November 27 2009

³<http://www.gnu.org/software/gdb/> - last accessed: November 27 2009

stored results. This analysis is done some time after a certain code snippet was executed and, therefore, does not serve a direct interaction.

8.2.1. Model-based Debugging

There exists a set of different concepts using models for debugging approaches [17]. Typically models are used for providing an understandable perspective to debug a certain programming language like Java in this context. The application that “outperforms” [17] all other presented approaches in the comparison of Mayer et al. [17] is MBSD (Model-based software debugging). It is an application providing Model-based Diagnosis (MBD) [24]. Its origin is set in the identification of “incorrect clauses in logic programs” [17], but has already added support for several other languages like Java. It creates a formal code representation – model – from the test case result. The focus of MBSD lies here on the identification of error origins instead of focusing on observing the code and its errors itself. The analysis itself takes place by calculating the deviation of the “observed behaviour” with “normal behaviour” [17]. According to Mayer et al. the most essential characteristic of MBSD is the role exchange of the model – reflecting “the behaviour of the (incorrect) program” – and the system – which specifies the “correct result” [17] in the test cases. A deeper view of usable methods and their relationships is described by Mayer et al. [16] al which is beyond the focus of the thesis.

8.2.2. QVT-R Debuggers

Beyond the general debugging principles and opportunities, very specific solutions can be identified for QVT-R. Such tools presented in this section offer similar functionalities as discussed for other languages and platforms before, but severely differ in their optimization for QVT-R.

Eclipse Debugger

The Eclipse Modeling Edition⁴ (Eclipse with Modeling Tools) is a freely available open source bundle of plugins for modeling and model transformation available in the Eclipse Modeling Project⁵ included in the basic Eclipse project. The relevant component for this thesis is the Model To Model (M2M) project that presents a solid ATL and an exemplary QVT Core support. There are still many open problems for the QVT Core execution, because the debugger is not integrated and not all concepts seem to be supported. At the time of writing OBEO⁶ is working on the integration of QVT-R functionality for M2M. As it is a rather new introduced field of the Eclipse project it can be expected that the QVT support will be improved steadily. It is planed to support all three branches of QVT in future versions. The Eclipse debugger that provides classical graphical debugging facilities. It allows stepwise execution of program codes controlled by breakpoints. So it can be directly compared with the basic functionality of

⁴Download: <http://www.eclipse.org/downloads/> - last accessed: November 27 2009

⁵<http://www.eclipse.org/modeling/> - last accessed: November 27 2009

⁶<http://www.obeo.fr/> - last accessed: November 27 2009

typical debuggers such as DDD, but provides deep IDE integration. However, the Eclipse debugger does not provide a graphical data visualization for which DDD is famous for. This debugging functionality is reused and configured for the language specific usage in related projects such as Medini QVT and many other top level applications. However, the unmodified Eclipse Debugger does not provide a good suitability for QVT-R.

Medini QVT

Medini QVT⁷ is a promising IDE for QVT Relations. It provides a powerful QVT engine, code validations, and a debugging perspective. The QVT Relations engine can be used to execute transformations and can, therefore, also be used for “Trial & Error”-mechanisms. The code validation helps to identify syntax errors. For example wrong spelled keywords are marked as incorrect and can be automatically corrected to existing ones. The debugging perspective offers a variable inspector. This view can show the value for each variable. This is the very same associated perspective like for other programming languages in Eclipse, because Medini QVT is based on Eclipse (Medini QVT is a set of plugins added to the base Eclipse project). So it offers a stepwise viewing of actual variable storages. Naturally, this is not enough to give an idea why a certain result is produced or how it is processed. This has to be tested by the developer and can be time consuming.

ModelMorf

The ModelMorf application⁸ supports the QVT Relations language. However, not all language constructs are already supported. In contrast to the other products discussed so far, ModelMorf is a commercial product.

ModelMorf positions itself as lightweight command-line model transformation application. It does not serve a graphical user interface, but, therefore, only uses about 7 Megabytes of hard disk space. However, the debugging functionality provided is unclear as no information could be found on this issue.

Summary

There is a multitude of other implementations available. However, projects like SmartQVT⁹ (QVT Operational) or Borland Together Architect¹⁰ (QVT Operational) have not added QVT Relations support yet and do not seem promising enough to add the support in the coming versions. ModelMorf, however, does support QVT-R, but no information about its debugging features could be found. Hence, only the previously discussed Debuggers are compared in this summarization.

⁷Download: <http://projects.ikv.de/qvt>-lastaccessed:November272009 - last accessed: November 27 2009

⁸http://www.tcs-trddc.com/solutions-software_rd.htm#ModelMorf - last accessed: November 27 2009

⁹<http://smartqvt.elibel.tm.fr/> - last accessed: November 27 2009

¹⁰<http://www.borland.com/us/products/together/> - last accessed: November 27 2009

8. Related Work

In the following Table 8.1 the most important properties for QVT Relations debuggers are presented. The table differentiates between “fully applies” (+), “partially applies” (~) and “does not apply” (-). The used properties are listed hereinafter: QVT Relations support – at the moment of writing – (QVT-R); Support for direct or indirect debugging (Debugging); Platform independent / Several platforms (Platforms); Graphical User Interface (GUI); and operational semantics visualization (Visualization).

| Application | <i>QVT R</i> | <i>Debugging</i> | <i>Platforms</i> | <i>GUI</i> | <i>Visualization</i> |
|-------------|--------------|------------------|------------------|------------|----------------------|
| Medini QVT | + | + | + | + | - |
| ModelMorf | ~ | - | - | - | - |
| Eclipse M2M | - | + | + | + | - |

Table 8.1.: Tabular summarization of QVT Relations tools

In this summarization the MediniQVT presents itself as most promising which results from its advanced integration of QVT Relations. All discussed products fall short in the category **Visualization** representing the visualization of the operational semantics. This means that this debugger provides rudimentary support for removing errors, but does not outstandingly assist in finding problem origins. Nevertheless, this point seems to be the most important for developers that have lost themselves in the complexity of a magnitude of possibilities resulting from complex bidirectional statements. Very often it is not obvious why a certain rule fires for a specific item.

8.2.3. Forensic Debugging Techniques

Another approach that can be used for debugging is forensic debugging. In particular interesting for the context of this thesis is the approach for forensic debugging of MTs introduced by Hibberd et al. [12]. In this research work a forensic approach is presented that tries to identify relationships between source elements, target elements and the referred MT rules. This approach uses trace models that can be seen as intermediary representations being able to answer debugging queries. These queries are used to identify problem areas and are assisted by program slicing mechanisms. In contrast to this approach, this thesis focuses on live debugging mechanisms are discussed and presented.

8.2.4. Backwards Debugging

The typical debugging procedure involves the starting at a certain (entry) point and step-by-step moving forwards to identify the problem area. However, there is another approach presented in the WHYLINE (“Workspace that Helps You

Link Instructions to Numbers and Events” [32]) system. It allows the querying of problem origins by asking questions that are answered by the debugger.

According to Zeller et al. [32] WHYLINE allows “Why did” and “Why didn’t” questions. The first type can be used to identify why a certain result is achieved for a specific line or element. Therefore, WHYLINE shows all elements backwards on which the variable or element is dependent on. The latter question can be used to “retrieve those statements that directly prevent execution. It then performs the why did questions on each” [32].

The user can also ask such informally called “why did”-questions [32] again at any point of the provided results again to better understand why a certain result was or was not achieved.

8.3. Verification

Not only classical debugging and testing approaches can be used for the error handling in QVT-R codes. There also exist other live and forensic verification approaches that mainly address the problems of observing codes and their errors (1), as well as the identification of error origins (2). In contrast to testing approaches verification tries to logical prove (verify) or analyze the code in lie of using test data or the intuition of the user. However, full verification of complex codes has a too high effort to be used.

8.3.1. Verification by Translation

The tooling support for direct debugging or verification of QVT Relations is very rare at the time of writing. Translational approaches try to overcome this issue by transferring the QVT Relations code in other textual representations that are better debuggable. For example the approach of Jouault and Kurtev [13] aims at executing the QVT-R mappings in the ATL Virtual Machine (ATL VM). This is achieved by a code transformation to an ATL representation. Moreover, there exists a concept of Romeikat et al. [26] to transform QVT-R to QVT-C code. The resulting statements can then be executed in the QVT-C implementation environments. Both approaches can help to improve the tooling support heavily, but both variants reduce the abstraction level. This leads to losing the easiness of realizing statements and as a result makes the overall process less efficient.

Another translational approach with similar ideas to the concept of Hibberd et al. [12] is the “graphical declarative debugger of incorrect answers for the constraint lazy functional-logic programming language TOY [1]” [5]. The debugging in a step-by-step manner introduced and used for many imperative languages is often “not suitable for debugging declarative programming languages” [5]. As already discussed QVT Relations is a declarative language and, therefore, faces similar problems. However, this debugger is based on the issues of functional languages like TOY. As visualized in Figure 8.3 the debugging process starts with the raw TOY code file. This file is syntactically and semantically analyzed and transformed to an intermediary code file. This resulting code is again checked by a type checker and afterwards generates prolog code. The last step is undertaken because this debugger is written in prolog. This debugger then obtains a formal

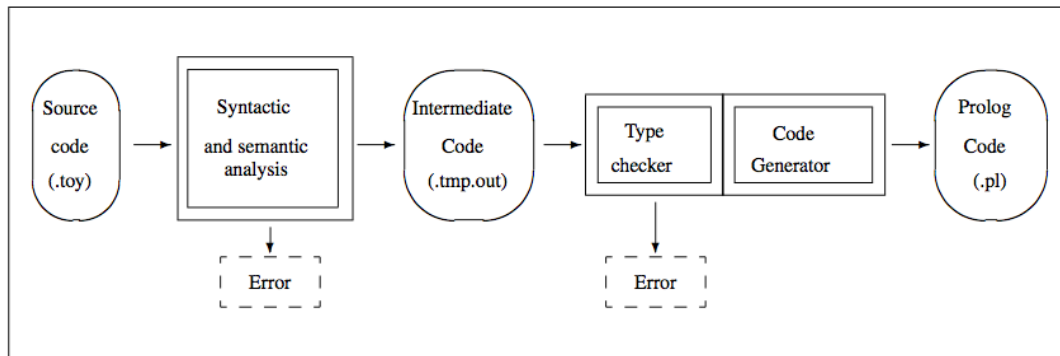


Figure 8.3.: Transformation of a TOY code file to a Prolog representation [5]

computation tree that is the entry point for debugging. From this point on the developer can freely inspect the results of the computation tree or call provided strategies “or finding out a buggy node and hence an incorrect program rule” [5].

8.3.2. Verification of Graph Transformations

As stated in [7] there is the possibility of representing models “as attributed, typed graphs” [7]. Because of this fact, graph transformation can be used for model transformation as well. Graph transformation rules consist of a left-hand-side (LHS) and a right-hand-side (RHS). Models are transformed to graphs. An example is visualized in Figure 8.4. The LHS and the respectively RHS of the models are mapped to its graph representations. The graph representations are more abstract and, therefore, allow better analysis of dependencies and relations. This process is supported by the graphical tool called AToM³ [9].

Graph transformations itself, however, need to be verified as well. There exists a verification approach provided by Zhao et al. [33] that tries to identify, if a design pattern still holds after evolutionary modifying its structures. The overall process starts by creating UML models representing the design patterns. Then the UML diagram as input information is modified by using a graph transformation engine and modification rules specified by the user to achieve a certain modification. Running the graph transformation results in a certain output pattern that needs verification.

“If a pattern can be reduced to an initial graph by a sequence of productions, the evolved pattern is considered to conform to the structural properties of a particular design pattern represented by the graph grammar and the evolution of the design is proved correct.” [33]

8.3.3. Verification based on CPN

Another mechanism is described by Juan de Lara and Esther Guerra in the paper “Formal Support for QVT-Relations with Coloured Petri Nets” [8]. This approach uses CPN to visualize QVT-R transformations. As this approach tries to visualize QVT-R in CPN, has some similar boundaries than the approach presented in this thesis. At the end of this section some open issues and differences

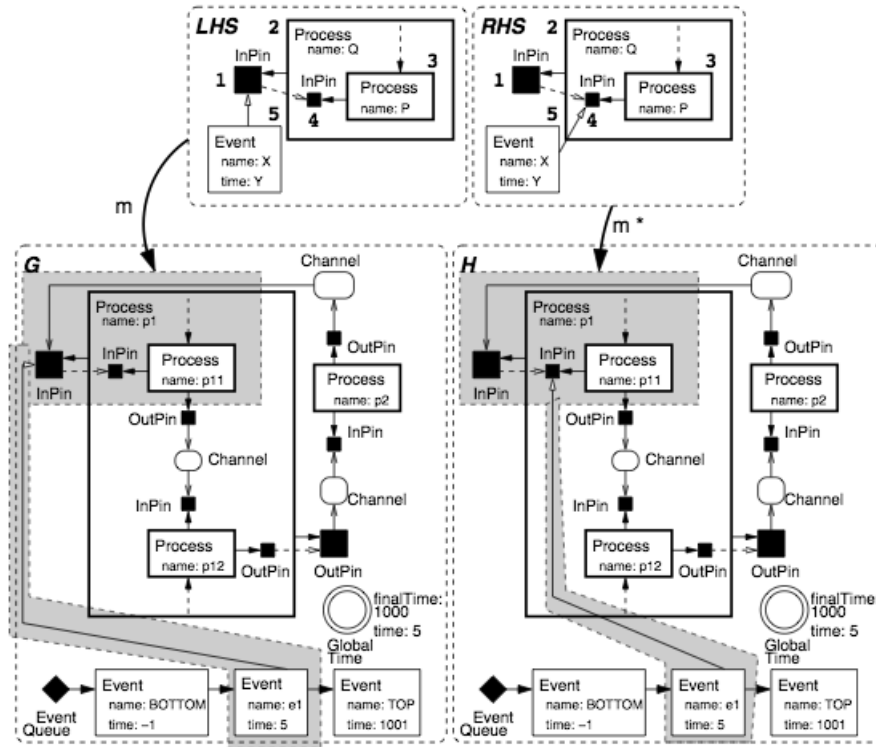


Figure 8.4.: A graph transformation rule [7]

to the own presented ideas are highlighted.

An overview of the visualization approach is given in Figure 8.5. In the center (the box with the green border) there exists a Relation with the name `PackageToSchema`. This Transition interacts with Places (holding the actual state) by using Arcs. The Arcs specify the direction from a certain source to a certain target model. On the right side of figure there is the definition of the actual data types, representing metamodel elements, e.g., `Package` or `Schema`.

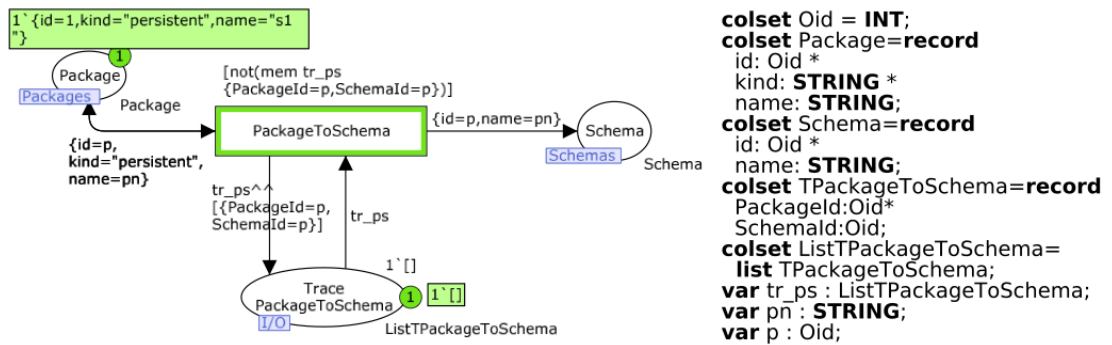


Figure 8.5.: Example CPN model: net (left) and colour set declarations (right) [8]

A more sophisticated generated example model in Figure 8.6 shows the usage of TracePlaces and more detailed interactions. The TracePlaces are needed to provide a possibility of passing values to other Relations as it is done with Where- and When-Clauses in QVT Relations. The Place `Trace PackageSchema` at the bottom of the figure represents such a TracePlace. Detailed specifications of the

8. Related Work

types in the Places are specified in the green pages close to them. For example ids or names are declared.

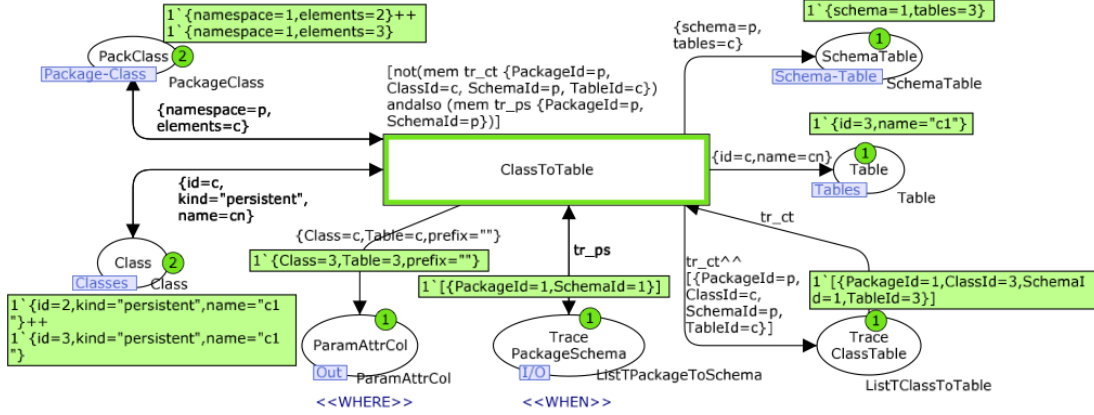


Figure 8.6.: Generated transition from relation ClassToTable [8]

Such representations only visualize Relations one-by-one. For a “high-level view” [8] another perspective is introduced (see Figure 8.7) using hierarchical Colored Petri Nets. This only describes the Relations and their interactions using the intermediary TracePlaces.

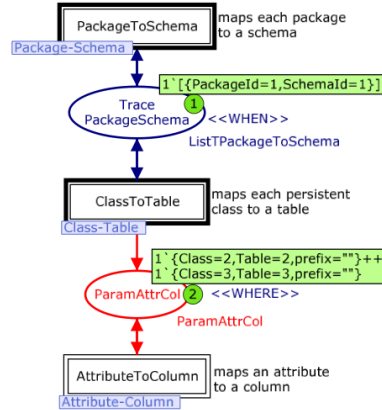


Figure 8.7.: A high granular view of the CPN model [8]

The approach presented in this paper provides a good modeling of QVT Relations mappings. Moreover, these models can be generated and analyzed. The usage of CPN creates the possibility of using CPNTools (cross-platform) and Petri Net analyzation tools. However, this concept does not try to fill the gap between declarations and its operational semantics for execution. For developers this gap can be an understanding barrier. In the approach presented in this thesis it is the main aim to visualize how a certain result could be achieved and all intermediary states. In this approach, however, only the dependencies of logical Relations and other QVT-R elements are visualized on a high-granular level. Therefore, it cannot be deducted in particular where the error occurs and for which state. In the approach based on TROPIC live interaction with particular objects from certain models should be accessible to highlight over which path objects are transformed and what why this has to happen.

8.4. Summary

As summarization the following Table 8.2 compares the discussed related work approaches using the hereinafter criteria: Automatable debugging functionality (**Automatable**); Observation of errors (**Observation**); Tracking of origins (**Origins**); Fixing of bugs (**Bugs**); QVT Relations support (at the time of writing) (**QVT-R**); Live debugging or testing (**Live**); Forensic debugging or testing (**Forensic**); Outstanding adequacy for identifying common pitfalls in QVT Relations coding (**Pitfalls**); and operational semantics visualization (**Visualization**). The discussed approaches represent the most promising alternative of this field to give an idea of the approach’s underlying potential.

| Approach | <i>Automatable</i> | <i>Observation</i> | <i>Origins</i> | <i>Bugs</i> | <i>QVT-R</i> | <i>Live</i> | <i>Forensic</i> | <i>Pitfalls</i> | <i>Visualization</i> |
|-----------------|---------------------------|---------------------------|-----------------------|--------------------|---------------------|--------------------|------------------------|------------------------|-----------------------------|
| Testing | + | + | ~ | - | ~ | + | - | - | - |
| Debugging | + | - | + | + | + | + ¹¹ | + ¹² | - | - |
| Verification | + | + | + | - | + | + | - | - | - |

Table 8.2.: Summarization of QVT Relations debugging approaches

In the comparison debugging with the respectable tool support (MediniQVT) turns out to be a well adjusted solution for classical error origin detection (or even error observation) in QVT Relations code. It lacks at the support for a visual representation of causal inferences (visualization of operational semantics) and does not exceedingly qualify for research in identifying typical pitfalls in the QVT Relations coding. Testing is a good approach for observing the source code and its errors, but is not applicable for fixing bugs or and only partially suitable for identifying the origins of errors. The verification approach, however, has typically a “higher distance” to the original code due to a more abstract methodology, but can provide good analysis opportunities. The identification of common pitfalls is not supported by any of these approaches in particular. However, there exists a machine learning [2] debugger for distributed system – proposed by Nitesh Chawla et al. [6] – that shows exceptional high potential for being used for the identification of common pitfalls. It focuses on forensic data analyzation and is itself, therefore, not appropriate for live origin tracking and bug removing.

Beyond the mentioned comparison of high level approaches it seems notable that the concept by de Lara and Guerra [8] – verification based on CPN – can provide a great improvement of the overall positioning of verification mechanisms. However, the tooling support itself is very unclear and the fact that the approach seems still in phase of conception makes it unrealistic to use for professional development at the moment of writing.

¹²Both live and forensic approaches are available and were discussed

8. *Related Work*

Although some interesting approaches or even tools have been discussed, some unsolved issues are still existing. No solution could be identified that can visualize the operational semantics behind a certain declaration. To identify errors in the QVT-R code a live mechanism is necessary that allows direct interaction and observation when which path is taken for execution for which objects of the model. Furthermore, it is necessary to analyze which correspondences and dependencies exist in the QVT-R code. There is no solution discussed here that can fully satisfy all these characteristics. Therefore, it is the aim of this thesis to provide such an approach taking the live interaction like described in QVT-R debuggers, combining them with the good dependency visualization of [8] and adding the ability to visualize all states and intermediary states.

9. Conclusion

This thesis is concluded with a summary of the presented content and an outlook on future work.

9.1. Summary

Models are intensively used in the Model Driven Engineering which have experienced a strong impetus over the last years. For automatically transforming models to structurally different models, model transformations (MTs) are used – in particular the model transformation language QVT Relations (QVT-R) is used. The MTs in QVT-R have to face the complexity resulting from the usage of very heterogenous source and target models. Additionally, QVT-R is a declarative approach tailored for bidirectional transformations. The declarative character also leads to the problem of impedance mismatch between design-time and runtime of MTs. This problem is tackled in this thesis by introducing a graphical debugging approach based on TROPIC.

The solution in this thesis is based on practical experiments leading to a conceptual approach (section 4) and a prototypical implementation (section 6). The conceptual approach for visualizing the operational semantics of QVT-R in TROPIC is considered to be the main contribution of this thesis.

The analysis of a QVT-R code starts with the classification of the Relations – the most essential elements of a QVT-R transformation. As described in section 4 this is accomplished by establishing a graph of Relations presenting a possible execution ordering. The principle focuses on transforming each Relation at once. Afterwards the next Relation is executed, if no preconditions avoids it. However, the concrete transformation mappings stating which conditions have to be fulfilled and which correspondences from source to target exist, have to take place in the execution of the Domains of each Relation. In these Domains a differentiation between executions on different execution levels has to take place (as stated in section 4). Another interesting discussed issue is the identification of correspondences between source and target types. In lieu of an incomplete heavyweight analyzation approach, a light-weight approach based on several assumptions and restrictions is used.

9.2. Future Work

The TROPIC Nets try to visualize many details hidden by the textual representation in QVT-R. This circumstance leads to the problem of scalability as the Nets tend to grow disproportionally. Therefore, it is a necessity to improve the pretty printing mechanism which has been out of the focus in this thesis.

As described in section 7 interpretations of QVT-R have been necessary, mainly corresponding to the realization of medini QVT. Future work could arise from the enforcement of other ways of interpreting the QVT-R code, e.g., adopting specific interpretations to future versions of the QVT-R specification. For this issue the usage of the stated assumptions reflecting the interpretations are problematic and, therefore, may involve future work for optimization or correction.

Moreover, QVT-R code compromises many OCL statements, e.g., for value assignments or variable usages. However, OCL expressions may consist of complex queries that involve several different types and their properties. Moreover, OCL supports functions like sorting or counting, the retrieving of specialized information, and the usage of programmatic constructs like loops and decisions. In this thesis, only simple expressions such as value assignments are supported. This is due to the fact, that the magnitude of language constructs in OCL and its complexity in nesting them, as well as the possibility of querying from different origins, will lead to an immense effort in visualization. Heavy use of complex OCL queries would lead to a magnitude of TROPIC elements on the Transformation Net. So, it might be necessary to transform an expression in a dedicated visualization, if this is requested by the user, but normally the complexity of queries should be hidden. An integration of OCL expression visualization, therefore, would be desirable.

The improvement of the identification of source and target element correspondences, furthermore, is essential as not all correspondences could be derived from the QVT-R code in this thesis. QVT-R itself does not specify correspondences on element to element level which has been addressed by an approach (described in section 4) that cannot handle complex type correspondences, if they are not declared by variables which are used on both source and target side.

Furthermore, specialized transformations for particular one-sided transformable primitive types could be necessary in the future. Grade allows to specify such dedicated transformations, but does not make use of them at the moment of writing. However, there can arise situations for which this is necessary.

The most inconvenient open task is the identification and integration of unsupported language construct combinations such as special cases or occasional usages. Although many theoretical considerations and practical experiments were undertaken, the conciseness of the QVT-R standard leaves enough space for interpretations leading to the usage of other unconsidered combination variants. In particular the most common QVT-R usages are documented, but the standard does not support a precise inference of how these declarations can be transferred to operational instructions which is attempted in this thesis.

A. Appendix

The source code of the Grade application presented in this thesis is glued in as DVD-Rom version.

The DVD-Rom includes the following items:

- The source code of the Grade application
- All plugins/components used by Grade
- The bundled Eclipse version used for Grade & TROPIC development
- A `Readme` file

Note for consumers of the digital edition: Please, visit the site of the Business Informatics Group (<http://www.big.tuwien.ac.at>) or the associated TROPIC project site (<http://www.modeltransformation.net>) for further information about this and related projects.

Bibliography

- [1] ABENGOZAR-CARNEROS, M., ARENAS-SANCHEZ, P., CABALLERO-ROLDAN, R., GIL-LUEZAS, A., GONZALEZ-MORENO, J., LEACH-ALBERT, J., LOPEZ-FRAGUAS, F., RODRIGUEZ-ARTALEJO, M., RUZ-ORTIZ, J., AND SANCHEZ-HERNANDEZ, J. TOY: A Multiparadigm Declarative Language. In *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming* (2004), ACM New York, pp. 329–334.
- [2] ALPAYDIN, E. *Introduction to Machine Learning*. MIT Press, 10 2004.
- [3] BEZIVIN, J. On the Unification Power of Models. In *Journal on Software and Systems Modeling* (2005).
- [4] BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. *Pattern-Oriented Software Architecture*, vol. 4 of *Software Design Patterns*. Wiley, 2007.
- [5] CABALLERO, R. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming* (2005), ACM New York, pp. 329–334.
- [6] CIESLAK, D., CHAWLA, N. V., AND THAIN, D. Troubleshooting thousands of jobs on production grids using data mining techniques. In *Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing* (2008), vol. 00, IEEE Computer Society.
- [7] DE LARA, J., AND GUERRA, E. Formal Support for Model Driven Development with Graph Transformation Techniques. In *Proceedings of the Congreso Español de Informática (CEDI'05)* (2005).
- [8] DE LARA, J., AND GUERRA, E. Formal Support for QVT-Relations with Coloured Petri Nets. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)* (2009), Springer Berlin / Heidelberg.
- [9] DE LARA, J., AND VANGHELUWE, H. ATOM³: A Tool for Multi-Formalism Modelling and Meta-Modelling. In *Proceedings of the Fundamental Approaches to Software Engineering (FASE'02)* (2002), Springer Berlin / Heidelberg.
- [10] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 11 1994.

- [11] GINER, P., AND PELECHANO, V. Proceedings of the 12th international conference on model driven engineering languages and systems (models'09). In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)* (2009), Springer Berlin / Heidelberg.
- [12] HIBBERD, M., LAWLEY, M., AND RAYMOND, K. Forensic Debugging of Model Transformations. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS'07)* (2007), Springer Berlin / Heidelberg.
- [13] JOUAULT, F., AND KURTEV, I. On the architectural alignment of atl and qvt. In *Proceedings of the 2006 ACM symposium on Applied computing* (2006), ACM, pp. 1188–1195.
- [14] KURTEV, I. State of the Art of QVT: A Model Transformation Language Standard. In *Proceedings of the Applications of Graph Transformations with Industrial Relevance: Third International Symposium (AGTIVE 2007)* (2007), Springer Berlin / Heidelberg.
- [15] KUSEL, A., RETSCHITZEGGER, W., SCHWINGER, W., AND WIMMER, M. Common Pitfalls of Using QVT Relations - Graphical Debugging as Remedy. In *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009)* (October 2009), IEEE Computer Society, pp. 329–334.
- [16] MAYER, W., AND STUMPTNER, M. Model-Based Debugging - State of the Art And Future Challenges. *Electronic Notes in Theoretical Computer Science* (2007).
- [17] MAYER, W., AND STUMPTNER, M. Evaluating Models for Model-Based Debugging. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)* (2008), IEEE Computer Society, pp. 128–137.
- [18] MÜSSELER, J. *Allgemeine Psychologie*, 2.0 ed. Spektrum Akademischer Verlag, 2008.
- [19] NOLTE, S. *QVT - Relations Language: Modellierung mit der Query Views Transformation*. Xpert.press. Springer-Verlag Berlin, 2009.
- [20] OBJECT MANAGEMENT GROUP. *Human-Usable Textual Notation (HUTN) Specification¹*, 1.0 ed., 8 2004.
- [21] OBJECT MANAGEMENT GROUP. *OCF 2.0 Specification²*, 2.0 ed., 06 2005.
- [22] OBJECT MANAGEMENT GROUP. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification³*, 1.0 ed., 4 2008.

¹<http://www.omg.org/cgi-bin/doc?formal/2004-08-01> - last accessed: November 27 2009

²<http://www.omg.org/cgi-bin/doc?ptc/2005-06-06> - last accessed: November 27 2009

³<http://www.omg.org/spec/QVT/1.0/> - last accessed: November 27 2009

- [23] OBJECT MANAGEMENT GROUP. The Architecture of Choice for a Changing World. <http://www.omg.org/mda/>, 2009.
- [24] REITER, R. *Artificial Intelligence*, vol. 32. 1987, ch. A theory of diagnosis from first principles.
- [25] REITER, T. *T.R.O.P.I.C.: Transformations On Petri Nets In Color*. PhD thesis, Johannes Kepler Universität, Institute of Bioinformatics, Altenberger Straße 69, A-4040 Linz, Austria, 2 2008.
- [26] ROMEIKAT, R., ROSER, S., MÜLLENDER, P., AND BAUER, B. Translation of QVT Relations into QVT Operational Mappings. In *Proceedings of the Second International Conference on Theory and Practice of Model Transformations (ICMT 2009)* (2009), Springer Berlin / Heidelberg.
- [27] RUSSELL, S., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*, 2. a. International Edition ed. Series in Artificial Intelligence. Prentice Hall International, 2003.
- [28] STEINBERG, D., BUDINSKY, F., PATERNOSTRO, M., AND MERKS, E. *EMF- Eclipse Modeling Framework*, 2.0 ed. the eclipse series. Pearson Education, Inc., 2009.
- [29] WIMMER, M., KUSEL, A., SCHÖNBÖCK, J., KAPPEL, G., RETSCHITZEGGER, W., AND SCHWINGER, W. A Petri Net based Debugging Environment for QVT Relations. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE 2009)* (2009), IEEE Computer Society, pp. 1–12.
- [30] WIMMER, M., KUSEL, A., SCHÖNBÖCK, J., KAPPEL, G., RETSCHITZEGGER, W., AND SCHWINGER, W. Proceedings of the 12th international conference on model driven engineering languages and systems (models’09). In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS’09)* (2009), Springer Berlin / Heidelberg.
- [31] WIMMER, M., KUSEL, A., SCHÖNBÖCK, J., REITER, T., RETSCHITZEGGER, W., AND SCHWINGER, W. Lets’s Play the Token Game – Model Transformations Powered By Transformation Nets. In *the International Workshop on Petri Nets and Software Engineering PNSE’09* (2009), pp. 35–50.
- [32] ZELLER, A. *Why Programs Fail - A Guide to Systematic Debugging*. Morgan Kaufmann and dpunkt.verlag, 2006.
- [33] ZHAO, C., KONG, J., AND ZHANG, K. Design Pattern Evolution and Verification Using Graph Transformation. In *Proceedings of the 40th Annual Hawaii International Conference on Systems Sciences (HICSS 2007)* (2007), IEEE Computer Society.

Bibliography

- [34] ZIMBARDO, P. G., GERRIG, R. J., AND GRAF, R. *Psychologie*. Pearson Studium, 2008.