Efficient and Compatible Bidirectional Formal Language Translators based on Extended Triple Graph Grammars

Vom Fachbereich 18
Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des Grades eines Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

vorgelegt von
Dipl.-Inform. Felix Klar
geboren in Frankfurt am Main

Referent:	Prof. Dr. rer. nat. Andy Schürr
Korreferent:	Prof. Dr. rer. nat. Albert Zündorf
Tag der Einreichung:	15.09.2011
Tag der Mündlichen Prüfung:	10.02.2012

D 17
Darmstadt 2012

Language is some biological property of human beings.

[Noam Chomsky]

in his talk about "Poverty of Stimulus" at JG University Mainz, 24.03.2010

For my parents,
Ursula
and
Reinhard

Abstract

In the context of model-driven engineering, models play an important role in everyday life. Models are used to abstract from certain subjects and to describe artifacts and procedures. In software engineering, a system under development is often modeled on different levels of abstraction and from multiple perspectives which results in plenty of models. Moreover, the resulting models depend on each other and the need for automatically translating between related models arises in order to reduce costs, errors, and laborious manual work and to speed-up development processes.

The model-driven engineering approach proposes model transformations as a key concept of model-based development which allows to automatically refine and transform models or translate between related models. Especially, bidirectional translators are often required which are able to automatically keep related models in a consistent state. The goal of bidirectional model transformations, which allow to execute transformations defined between a source and target model in both directions, is to assist in such situations. To be able to specify (bidirectional) model transformations the need for (bidirectional) model transformation languages arises.

Triple graph grammars (TGGs) are a formally founded bidirectional transformation language based on graph transformations with precisely defined semantics. A TGG specification describes correspondence relationships between two languages and consists of a set of productions that declaratively specify the simultaneous evolution of both related models. The main advantage of triple graph grammars is the possibility to automatically derive bidirectionally working forward and backward translators from a TGG specification that fulfill the fundamental properties efficiency and compatibility.

The grand challenge is to build translators that are efficient on the one hand and are compatible with respect to the TGG specification on the other hand. Compatibility means that translators are correct and complete with respect to the specification, i.e., pairs of models are in a consistent state after the translation operation and valid models are able to be translated into corresponding models. Moreover, the overall expressiveness of the triple graph grammar language has to be increased in order to create usable transformation specifications. But, it has to be ensured that derived translators still fulfill the fundamental properties.

In this thesis, the expressiveness of triple graph grammars is increased by supporting negative application conditions (NACs) that allow to restrict the applicability of transformation rules, which is required in certain cases. In addition, we accept the challenge of providing an efficiently working translation algorithm that still fulfills the properties correctness and completeness. We extend the expressiveness of triple graph grammars by a precisely defined class of NACs together with a new translation algorithm such that for the first time the fundamental properties of TGG-based translators are still satisfied. The resulting translators nevertheless have a polynomial runtime complexity and, therefore, can be considered efficient. Moreover, they are compatible with their triple graph grammar, which makes these translators usable in practice. In conclusion, the extended triple graph grammar formalism is applicable in real world scenarios, where model transformations are bidirectionally executed to keep related models in a consistent state.

Zusammenfassung

Im Zusammenhang mit der modellgetriebenen Entwicklung spielen Modelle eine wichtige Rolle im täglichen Leben. Modelle werden verwendet um von bestimmten Gegebenheiten zu abstrahieren und um Artefakte und Abläufe zu beschreiben. In der Software-Entwicklung wird ein zu entwickelndes System zumeist auf unterschiedlichen Abstraktionsebenen und von mehreren Perspektiven aus betrachtet, was in einer Vielzahl von Modellen resultiert. Die daraus resultierenden Modelle hängen voneinander ab und eine automatische Übersetzung zwischen diesen Modellen ist wünschenswert, um Kosten, Fehler und mühsame Handarbeiten zu reduzieren und Entwicklungsprozesse zu beschleunigen.

Der modellgetriebene Entwicklungsansatz schlägt Modelltransformationen als ein Schlüsselkonzept der modellbasierten Entwicklung vor, das es erlaubt automatisch Modelle zu verfeinern und zu transformieren oder zwischen zueinander in Beziehung stehenden Modellen zu übersetzen. Insbesondere werden oft bidirektionale Übersetzer benötigt, die in der Lage sind in Beziehung stehende Modelle automatisch in einem konsistenten Zustand zu halten. Das Ziel von bidirektionalen Modelltransformationen ist es, in den genannten Situationen zu unterstützen. Bidirektionale Transformationen erlauben es, die zwischen einem Quell- und Zielmodell definierten Transformationen in beide Richtungen auszuführen. Um (bidirektionale) Modelltransformationen zu spezifizieren, werden (bidirektionale) Modelltransformationssprachen benötigt.

Tripel-Graph-Grammatiken (TGGs) sind eine formal fundierte bidirektionale Transformationssprache, die auf Graphtransformationen beruht und eine präzise definierte Semantik besitzt. Eine TGG-Spezifikation gibt die übereinstimmenden Zusammenhänge zwischen zwei Sprachen an und besteht aus einem Satz von Produktionen, die in deklarativer Weise die simultane Entwicklung zweier in Beziehung stehender Modelle beschreibt. Ein Hauptvorzug von Tripel-Graph-Grammatiken ist die Möglichkeit automatisch bidirektional arbeitende Vorwärts- und Rückwärtsübersetzer aus einer TGG-Spezifikation abzuleiten, die die fundamentalen Eigenschaften "Effizienz" und "Kompatibilität" erfüllen.

Die große Herausforderung ist es Übersetzer zu bauen, die auf der einen Seite effizient sind und auf der anderen Seite kompatibel bezogen auf ihre TGG-Spezifikation. Kompatibilität bedeutet, dass Übersetzer korrekt und vollständig bezogen auf ihre Spezifikation sind, d.h. Paare von Modellen sind nach dem Übersetzungsprozess in einem konsistenten Zustand und Modelle, die gültig gemäß Spezifikation sind, können in entsprechende Modelle übersetzt werden. Zudem wird die Ausdrucksstärke von Tripel-Graph-Grammatiken erhöht, um in der Lage zu sein benutzbare Transformations-Spezifikationen zu erstellen. Dabei muss immer darauf geachtet werden, dass abgeleitete Übersetzer die oben genannten fundamentalen Eigenschaften erfüllen.

In dieser Arbeit wird die Ausdrucksstärke von Tripel-Graph-Grammatiken durch negative Anwendungsbedingungen (NACs) erhöht. Diese erlauben es die Anwendbarkeit von Transformationsregeln einzuschränken, was in bestimmten Fällen nötig ist. Zusätzlich stellen wir uns der Herausforderung einen effizient arbeitenden Übersetzungs-Algorithmus zu entwickeln, der zusätzlich die Eigenschaften "Korrektheit" und "Vollständigkeit" erfüllt. Wir erhöhen die Ausdrucksstärke von Tripel-Graph-Grammatiken um eine präzise definierte Klasse von NACs und stellen einen neuen effizienten Übersetzungs-Algorithmus vor, so dass gleichzeitig die fundamentalen Eigenschaften von TGG-basierenden Übersetzern erfüllt bleiben. Die resultierenden

Übersetzer haben trotzdem polynomielle Laufzeitkomplexität und können daher als effizient angesehen werden. Desweiteren sind sie kompatibel mit ihrer Tripel-Graph-Grammatik, was diese Übersetzer in der Praxis einsetzbar macht. Zusammenfassend lässt sich sagen, dass der erweiterte Tripel-Graph-Grammatik-Formalismus in Alltagsszenarien einsetzbar ist, in denen Modelltransformationen bidirektional ausgeführt werden, um miteinander in Beziehung stehende Modelle in einem konsistenten Zustand zu halten.

Contents

List of Figures

List of Tables

1. Introduction

This chapter gives an introduction to this thesis. We start in Sect. 1.1 with a short motivation of model-based development and model translation. Then, Sect. 1.2 more precisely defines the scope of this thesis. Section 1.3 discusses the state of the art when this thesis was started. The contributions of this thesis are presented in Sect. 1.4. Finally, Sect. 1.5 gives an outline of this thesis and gives a suggestion of how to read this thesis.

1.1. Motivation

Nowadays, computers are widely used to assist their users in different *domains* of the real world. Amongst other things computers store and manipulate data, perform calculations, automate and simulate complex processes, facilitate communication and display information in various ways. Domains that benefit from the usage of computers are, e.g., finance, banking, sports, entertainment, astronomy, mechanical engineering, and software engineering. In order to be computable, the artifacts and procedures of these domains are mapped—as models—to the world of computers. Such models are stored inside a computer as *structured data* which complies with a formal language. A model conforms to a specific language which specifies the syntax and semantics of the model's elements so the meaning of the model becomes clear to a viewer. Specific domains of the world require specific languages which are nowadays often called *domain-specific language (DSL)s* [FP10]. These DSLs are tailored to the specific needs of a group of people that are involved in the execution of specific tasks in a given domain.

Let us consider the software engineering domain where one task is to build software systems. To achieve this goal common development processes like the V-model, Rational Unified Process, or XP are typically applied. These development processes divide the whole process into a series of operations which are called *activities*, *steps*, and *phases* respectively. According to [Som07] and [JBR99] these activities are software specification (i.e., requirements analysis), software design, implementation, testing, deployment, and maintenance. Depending on the underlying specific development process the activities are executed iteratively in a defined order until the realization of the software system is finally completed. Different activities produce models on different levels of abstraction. Early activities yield more abstract models, whereas later activities typically produce more concrete models. In particular, textual or graphical models, like textual requirement documents and graphical system interrelationship documents are created, edited, and then passed to other activities, where they are enriched and refined or used as foundation for new models. Each activity represents a specialized (sub)domain that has its own *domain experts* which are responsible for different aspects of the system. In most cases, these *subdomains* have a non-empty intersection. Consequently, there are DSLs for these subdomains that share the same artifacts of the real world and of

the software system. These DSLs may differ in their representation of artifacts only, or they regard additional aspects that are not relevant in the other subdomain. Moreover, competing DSLs of different organizations may exist that represent the same domain but, e.g., use different naming conventions and have a different language design. Hence, model elements in the DSLs of specific subdomains are often related to each other.

An example for different representations of models that share the same domain are use cases that are created during requirements analysis. For example, use cases are represented as UML use case that depicts actors, use cases, and their relationships graphically, or they are represented textually based on a set of templates (cf. Fig. 1.1). Graphical use cases are used in a rough overview of the system, whereas textual use cases contain more detailed information. Intersecting subdomains that are related to each other are implementation and testing, as each implemented feature of a software system has tests that cover the implementation in order to ensure the quality of the implementation.

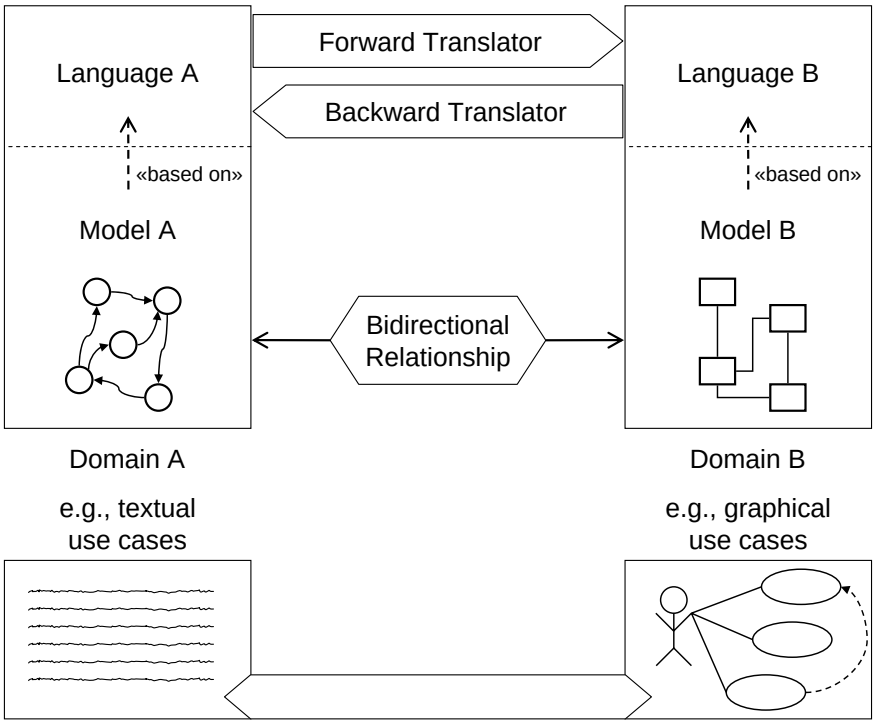


Figure 1.1.: Overview of bidirectional model translators.

DSLs typically come with tool support and, therefore, many *tools* are used throughout a software engineering process. Each tool has its own data structures due to the DSL it implements and is applicable in its specialized domain. In general these tools come from different vendors and are heterogeneous regarding their import/export capabilities. Therefore, it is a quite common scenario that models managed by one tool are translated into models of another tool manually to offer domain experts the information required to perform their tasks. In addition, iterative development processes cause activities to be executed over and over again. One of the worst cases in software engineering is to detect flaws in the requirements during

advanced phases of the design and implementation activities. In this case many models are affected and have to be reviewed when eliminating the flaw in order to remove inconsistencies that arise among related model elements.

Due to these facts the need for (semi-)automatic *model translators* emerges. The need for translators is not restricted to the domain of software engineering. Instead, translators are more generally required in various other domains mentioned earlier. Translators should be able to translate instances of one language into another language, detect inconsistencies and perform incremental updates after changes occurred in a model. In order to keep related models in a consistent state, *bidirectional* model translators are required. These translators have no fixed order of creating and changing interrelated model elements and are able to translate forward and backward between models that depend on each other. Typically, a bidirectional relationship is specified between the elements of two related languages from which forward and backward translators are derived (cf. Fig. 1.1). Models based on one language are then translatable into the corresponding language and vice versa.

The grand challenge is to build bidirectional translators that are efficient and compatible with respect to their specification. Moreover, the specification language needs to provide enough language features such that the expressiveness of the language has the capability to create specifications that are applicable in practice. The compatibility property consists of the two properties *correctness* and *completeness*. A translator is correct if it creates pairs of related models that are in a consistent state with respect to the specified model consistency relation. Completeness is fulfilled if every model that is an element of the language defined by the specification is translatable into its corresponding representation in the related language. The mentioned properties compete against each other. Especially, it is hard to guarantee efficiency in conjunction with completeness. In the following chapters we will accept the challenge for providing an expressive language for creating bidirectional translators that fulfill the properties efficiency and compatibility.

1.2. Scope

In the context of model-driven engineering and the *Model Driven Architecture* (MDA) [KWB03] the Object Management Group (OMG) proposes to build software systems based on models using modeling languages like UML and MOF and the standardized *model transformation language* QVT [Obj08]. An overview to model transformations is given by [SK03]. A classification of model transformation is presented by Czarnecky and Helsen in [CH03] and [CH06]. The taxonomy of model transformation is discussed by Mens et al. in [MCG05]. There is an active community behind model transformations which discusses topics of interest in the International Conference on Model Transformation (ICMT)[1]. According to the ICMT, topics of interest include, but are not limited to languages, scalability, reuse, semantics, implementation, generation, merging, maintenance, evolution, methodologies, tools, case studies of/for model transformations.

A model is typically transformed by applying a set of transformation rules. A rule consists of a matching part that finds a given situation in the model and a transformation part that

[1] `http://www.model-transformation.org/`

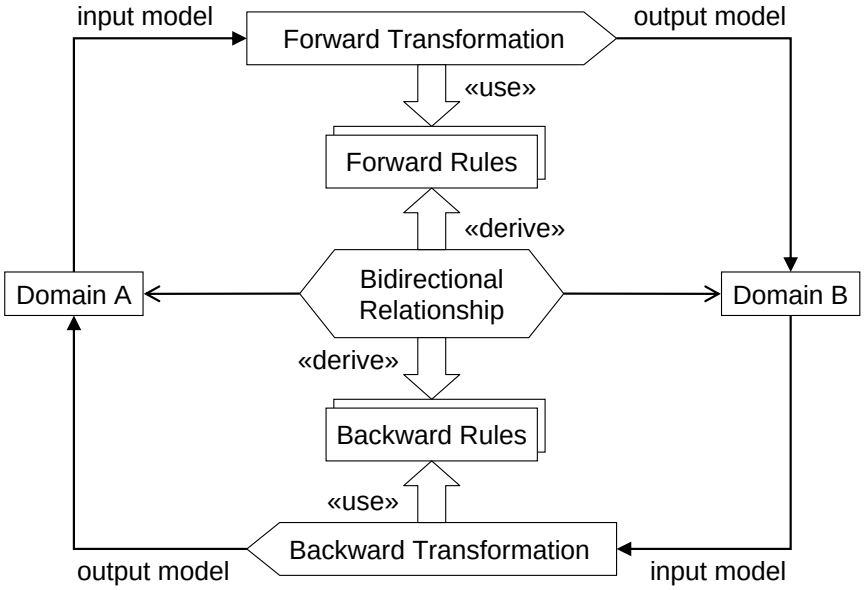


Figure 1.2.: Overview of bidirectional transformations based on relations.

modifies the model according to the rule semantics. A transformation rule is either *unidirectional* which means the rule is able to transform one-way only and another rule is needed that specifies the way back, or *bidirectional* which means the rule is applicable *forward* and *backward*. Translations as discussed in the preceding section are a special kind of model transformation that translate from one (domain-specific) language into another[2]. One approach for building translators based on relations is to derive forward and backward rules from a bidirectional relation specification, which are then utilized by forward and backward transformations (cf. Fig. 1.2).

The MDA approach coincides with our previously discussed perception that models are a vital part of different processes and that models should be transformed automatically. To build computer aided translators for formal languages it seems rather consequent to also develop such translators in a model-based way. So, the advantages of model-based development, e.g., increased productivity and quality, are directly transferable to the model-based building of translators.

In research activities of the Real-Time Systems Lab at the Technische Universität Darmstadt the metamodeling language *MOSL* (MOFLON Specification Language) has been created (cf. [Ame09] and [Kön09]). By means of MOSL it is possible to develop formal languages and the mentioned bidirectional translators in a model-based way. MOSL consists of the standardized languages MOF [Obj06] and OCL [Obj10b] and the well-known graph grammar based languages SDM [FNTZ00] and TGG [Sch95] (cf. Fig. 1.3). MOF and OCL are used to specify the structure and the static semantics of the language under development. SDM (Story Driven Modeling) is used to manipulate language instances and to define their dynamic semantics by means of unidirectional model transformations. TGGs (Triple Graph Grammars) are used to relate corresponding elements of two languages. Therefore, bidirectional transformation rules,

[2]In contrast, transformations in general do not necessarily change the language of the resulting model.

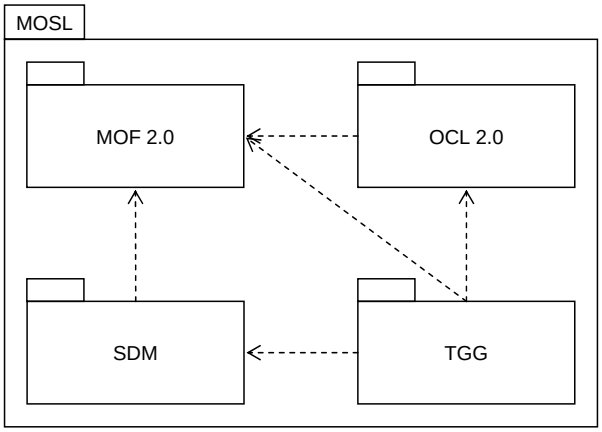


Figure 1.3.: Composition of the MOFLON Specification Language (MOSL) (from [Kön09]).

so-called *TGG productions*, are created in a TGG specification. Translators for both directions are then derived from a TGG specification. SDM and TGGs realize model transformations and translations. Both SDM and TGG are languages based on graph grammars and graph replacement systems which are formally founded and well studied [EEKR97, EEKR99].

Within the scope of this thesis model translators based on these technologies are studied and new concepts are developed to extend current bidirectional model translation approaches. We have chosen to use the well-known mapping of class diagrams and relational database schemata as running example. The extensions are evaluated in this running example.

1.3. State of the Art

When this thesis was started in 2006, bidirectional model transformations based on triple graph grammars became of interest in many research labs. This becomes visible when looking at the number of publications that were produced at this time in the context of TGGs[3]. In [SK08] we have stated some fundamental problems that were still unsolved in 2008 when this publication was written:

1. TGGs have been invented to specify mappings between two languages of graphs, but most published approaches either use inefficient graph grammar parsing and/or backtracking algorithms or rely on not very well-defined constraints of processed TGGs such that they are not able to guarantee important properties of derived forward and backward graph translations (FGTs/BGTs) with their TGG.

2. In practice urgently needed negative application conditions (NACs) of TGG productions are either simply excluded or handled in a way that destroys the fundamental properties of TGGs, i.e. derived FGTs/BGTs may generate graph triples that can't be generated by the original TGG.

[3] cf. `http://www.es.tu-darmstadt.de/english/research/projects/tgg/publications/` for a more or less complete survey of TGG publications published between 1994 and 2008.

3. Finally, appropriate means for modularization, refinement, and reuse of TGGs have not been studied until recently despite of the fact that quite large TGG specifications have already been created and used in industrial case studies.

As already mentioned, important properties of translators are *correctness* and *completeness* with respect to a TGG specification and *efficiency* of translators by means of space and time consumption. Translators presented so far refrain from providing efficient translators that also fulfill the completeness property. Moreover NACs are not handled in an appropriate way such that translators fulfill the correctness property. The overall goal at TGG language level is to provide certain language features such that the expressiveness of TGGs is increased in such a way that it becomes usable in practice.

The work of Königs [Kön09], which directly precedes the work done in this thesis, provides an algorithm that defines correct translators, but has an exponential worst-case runtime behavior, i.e., is not efficient.

In 2009, Ehrig et al. started addressing problems 1 and 2 listed above from a formal point of view in [EHS09] and [EEHP09]. Efficiency was out of scope in these publications. But, Ehrig et al. recently also addressed the efficiency property of translators in [HEGO10]. However, they discuss solutions on a very formal level and provide concepts that may be implemented by tool developers but do not provide concrete algorithm implementations applicable by translators.

In [SK08], [KLKS10], and in this thesis we focus on problems 1 and 2 listed above. Problem 3 was addressed by us in [KKS07] but is out of scope in this thesis and is up to future work.

1.4. Contributions

This thesis is based upon the original triple graph grammar approach presented in [Sch95]. Moreover, this thesis is an extended version of [KLKS10] which in turn is based on ideas firstly presented in [SK08]. The work presented in this thesis contributes to the meta-CASE tool MOFLON [AKRS06] and publications based thereon, e.g., [Kön09], [Ame09], [AKK+08], [KRS09], and [KS06].

This thesis presents a formalism based on triple graph grammars that can be utilized to derive bidirectional working language translators. Therefore, the triple graph grammar formalism originally presented in [Sch95] is extended by a certain class of negative application conditions to increase its expressiveness. We present for the first time a translation algorithm for translators based on triple graph grammars that is correct and complete with respect to a TGG specification and efficient in terms of space and time consumption. Moreover, the code generators of the MOFLON project have been adjusted for generating more sophisticated translation implementations.

The main results of this thesis are as follows:

- We introduce a well-defined class of negative application conditions for restricting the applicability of TGG productions. As a consequence, supported TGG productions must have the property that they are *integrity-preserving*, i.e., the graph that results from a translation satisfies a set of constraints after production application.

- We propose a handling of NACs when deriving forward and backward translators from a TGG specification. We found that NACs can be safely removed from the input component of derived translation rules under certain conditions.
- We propose *local completeness criteria* for triple graph grammars. These criteria, which affect either the source domain or the target domain of a TGG production, are used to prove the completeness property of translators derived from a TGG specification.
- We introduce a so-called *dangling edge condition* for triple graph grammars. This condition causes a transformation rule to be only applied if no dangling edges are produced in the input graph given to a translator. The dangling edge condition affects the efficiency of translators derived from a TGG specification because rule applications are prevented that would produce an incomplete translation due to the calculation of a sequence of translation steps that results in a dead-end and would require backtracking.
- We propose an advanced graph translation algorithm based on extended triple graph grammars. We prove that translators which apply this algorithm are efficient and compatible with respect to their TGG specification.
- The presented algorithm is designed to handle TGG productions that create more than one model element in source and target domain. This situation frequently occurs in typical TGG productions and, therefore, should be supported by TGG-based algorithms.
- We present a mapping from models to graphs that is used to demonstrate *how* graphs can be used as formal basis for model-driven purposes.

Based on the main results it is now possible to specify bidirectional relationships between related domain-specific languages with an extended triple graph grammar formalism. The translators derived from such a specification are correct and complete with respect to the specification and can be executed efficiently.

1.5. Outline

This thesis is structured as follows. Chapter 2 explains the fundamental concepts used throughout this contribution. Afterwards, application domains of the approach presented in this thesis are discussed in Chap. 3. There, fundamental properties that are important when building translators that are based on formal bidirectional transformation languages are discussed. The triple graph grammar language used to realize the mentioned translators is then presented in Chap. 4, where its main features and current limitations are discussed. Afterwards, the triple graph grammar approach is extended due to reasons of expressiveness in Chap. 5 to make it more usable in practice and to improve its acceptance. Especially, we formally extend the TGG approach by a certain class of negative application conditions in Sect. 5.2. Moreover, we demand that TGG specifications fulfill a local completeness criterion in Sect. 5.2.4 and that translators only perform a transformation step if a dangling edge condition is satisfied (cf. Sect. 5.3). Chapter 6 then presents a new efficient translation algorithm

utilized by forward and backward translators derived from TGG specifications that is correct and complete with respect to the TGG specification. The implementation of the extended approach in the meta-CASE tool MOFLON is described in Chap. 7. Chapter 8 compares our approach with related work. Finally, Chap. 9 concludes this thesis and gives suggestions for future work.

We suggest reading this thesis in a sequential order starting in Chap. 2, where the fundamentals of this thesis are discussed. Especially, Sects. 2.5, 2.6, and 2.7 that discuss model transformations in general, graphs, graph grammars and graph transformation are of interest. Then, Sect. 2.8 presents our mapping of models to graphs, which explains how models are realizable by graphs. The fundamental chapter closes in Sect. 2.8.3 by discussing how model transformations are realized with the story driven modeling language, which is based on graph transformations. Readers familiar with these fundamental issues may, however, skip the fundamentals chapter. A natural language translation analogy is discussed in Sects. 3.1 and 3.2, which is optional but informative. Likewise, Sect. 3.4 which presents the similarities in natural and formal language translation is optional reading. The discussion of the running example of mapping class diagrams and database schemata onto each other is started in Sect. 3.3. Readers familiar with this example may quickly review this section. Sections 3.5 and 3.6, which close Chap. 3 are mandatory. Chapter 4 is optional for readers familiar with triple graph grammars, but absolutely required for readers not familiar with triple graph grammars. However, Sect. 4.7, where the fundamental properties of TGGs and translators are discussed, is mandatory reading even for TGG experts. Chapters 5 and 6—more precisely Sects. 5.2, 5.3, 6.6, 6.8—contain the main results of this thesis and, therefore, are mandatory. Readers that are interested in the implementation of the approach discussed in this thesis are encouraged to read Chap.7. Chapter 8, which contains an overview of related work, is dedicated to readers that like to compare the approach discussed in this thesis with related approaches.

Readers that only want to get a quick overview of this thesis are encouraged to read Chap. 1 and Sect. 9.1.

Interested readers may find some auxiliary material in the appendices. Appendix A contains a glossary of the most important terms related to (triple) graph grammars and derived translators. Appendix B hints at some issues in the extended version [Sch94] of the original TGG publication [Sch95].

2. Fundamentals

The main goal of software engineers is to develop software systems—applying software engineering techniques. On the one hand, the level of abstraction of involved artifacts must be high enough so the engineer can effectively perform his tasks. On the other hand, the artifacts must be formal enough so computers are able to process them. There are many different approaches that have been developed during the last decades. Starting with assembler languages, continuing to higher level programming languages like FORTRAN, Java, C++, and many others, up to model-based engineering techniques applied today.

In this chapter we give an overview of the fundamental approaches and techniques applied in this thesis. We relate the terms "language", "graph", and "domain" to the context of model-driven (software) engineering (MDE) which is the model-based approach to software engineering. Then, we will present some modeling languages used by software engineers today. Subsequently, we will explain how these modeling languages are applicable in the modeling framework *Model Driven Architecture* (MDA). Afterwards, we point out the importance of model transformation in the MDE context. Finally, we introduce graph grammars, a formal approach that is applicable to realize model transformation.

2.1. Domains

The term "*domain*" is comprehensive and used in different fields of activity which include biology, mathematics, physics, and information technology. In a broad sense a domain is

> *[...] an area of knowledge or activity [...]*
>
> [Merriam-Webster's Dictionary [Per08]]

> *[...] an area of activity or knowledge [...]* ORIGIN *Old French* demeine *'belonging to a lord'.*
>
> [Oxford English Dictionary [SHE06]]

In the context of this thesis, we understand a domain, more precisely *problem domain*, as an application area that demands software to solve a problem. To build a software system that assists solving problems in a particular domain, common software engineering principles, methods, and processes are applied [Som07]. In practice many software systems are built that share the same problem domain. In order to facilitate reuse of common parts of these similar systems a software engineering concept named *domain analysis* has proven to be helpful. This concept was introduced by Neighbors [Nei80]. A domain analysis produces *domain models*

which consist of the elements[1] and operations of the activities that are specific to the domain. A domain model is a key feature to enable software reuse [dCLF93] and to produce systems related to the regarded application area rather efficiently. According to [PD90], *domain experts* and *domain analysts* extract relevant information from different sources of domain knowledge and analyze and abstract it. Then, *domain engineers* organize and encapsulate the results in the form of domain models, standards, and collections of reusable components. Domain models range from domain taxonomy to functional models, domain languages, and standards. The latter may include requirements specifications and design methods, coding standards, development procedures, management policies, and maintenance procedures for libraries containing reusable components.

Examples of problem domains are:

traffic management How do vehicles get through the streets without causing a traffic jam?

sales A customer has purchased an item with his credit card. How does the vendor get his money?

reservation systems A person wants to go to holiday. Which hotel is appropriate and how to get there?

library management A public libary offers to borrow books. Which documents of a particular author are available? Which documents are related to a particular topic?

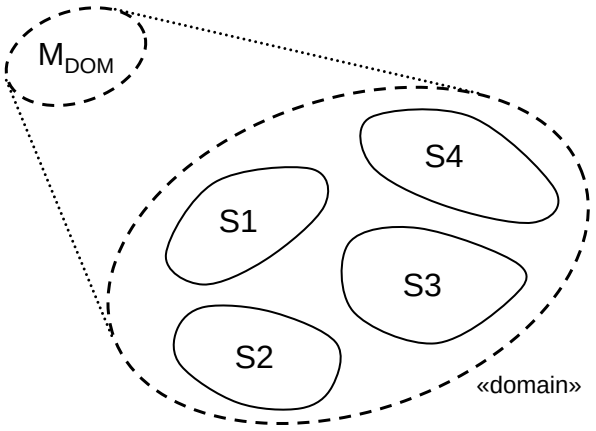


Figure 2.1.: A domain with domain model M_{DOM} and systems $S1$ to $S4$.

Figure 2.1 depicts the situation, where a domain analysis has produced the domain model M_{DOM} of a domain, e.g., the library management domain. The elements of the domain model are, e.g., stock, books, authors, and topics. The operations are *query stock by given author* and *query stock by given topic*. Based on this model four library systems $S1$ to $S4$ are created, where $S1$ is, e.g., the *Universitäts- und Landesbibliothek Darmstadt* (ULB) and

[1] In [Nei80] the term "object" is used. To avoid clashes with other usages of "object" later on in this thesis, we use the equivalent term "element"

$S2$ the *Universitätsbibliothek Kassel* (UBK). The library systems are then used to solve the problems in the library management domain, e.g., by the query: *query stock of* $S1$ *where topic = "software engineering"*. These two library systems are probably not just two different instances of the same software system, but represent two different incarnations of a family of software products that share a number of features. Moreover, they also offer functions that are specific for a given location, i.e., ULB and UBK respectively.

Another example of a problem domain is *software engineering*. The main problem of the software engineering domain could be stated as *"How to build and maintain a software system?"*. One might argue that software engineering is a *meta domain* which produces software solutions that are used to solve problems in other domains (cf. term "meta" in Sect. 2.2.5).

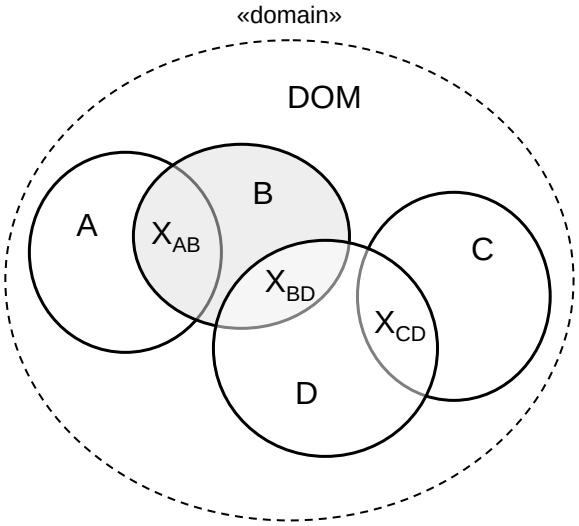


Figure 2.2.: Domain *DOM* with subdomains A to D.

Figure 2.2 depicts a domain *DOM* that consists of a number of *subdomains* A, B, C, and D. A subdomain contributes to solve the problems existing in the domain *DOM*. The problems regarded in a subdomain are focussed on a subset of all the problems that arise in the superdomain *DOM*. Some of the subdomains intersect (X_{AB}, X_{BD}, and X_{CD}), i.e., they share a subset of concepts. Hence, they share elements and operations.

Applied to the software engineering domain, *requirements analysis*, *software development processes*, and *tool integration* are subdomains. All of them are related to the problem of building software systems. Requirements analysis handles problems related to determining the requirements of a software system. The domain of software development processes is concerned with the problem of managing the workflows and activities while building a software system. Finally, tool integration handles the problems that arise when integrating existing software systems (i.e., tools) into a new software system. Revisiting Fig. 2.1, a domain model is produced for each of the subdomains during domain analysis. The domain models are then used to create systems for the specific problem domains. In case of the subdomain *requirements analysis*, this would result in building different software systems that are applicable by requirements analysts. In the more general case of the domain *software engineering*, tools are built that assist software engineers in building other tools.

2.2. Models and Languages

Models are widely used ever since in different areas and nowadays gain more and more attention in software engineering. In this section we will discuss what models are, how they are represented, and how models are related to each other and to languages. In addition, the term "metamodel", its relation to languages and metalanguages is discussed, and its usage in the context of this thesis. These discussions are performed on an informal level. A formal definition of the terms "model" and "metamodel" is out of scope and is still a topic of ongoing discussion. The interested reader is referred to a recent contribution on this issue by Kühne [Küh06b] and a discussion that originated from this contribution between Hesse [Hes06] and Kühne [Küh06a].

2.2.1. Models

Let us have a look at two definitions[2] of the term "*model*" in the MDE context.

> *A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made.*
>
> [Kühne [Küh06b]]

> *A model is a simplification of a system built with an intended goal in mind [...].*
>
> [Bézivin and Gerbé [BG01]]

In these definitions the term "system" is used as synonym for *subject of a model* and *original of a model*. Moreover, this includes both *systems to be built in the solution domain* and *systems to be described in the problem domain* [Küh06a]. So, software systems as discussed in Sect. 2.1 are also part of these definitions. But, the definitions are not limited to such systems. The definitions also include that another model may play the role of the system (i.e, subject). In this case, a more abstract model of a model is created. We will see later on in this section that there are different kinds of inter-model-relationships.

The previous definitions indicate that a model represents real or language-based subjects[3]. Furthermore, they show that a model is no copy of a subject. A copy does not abstract from details. Contrary, a model represents the subject on a more abstract or simplified level. This of course results in loss of details compared to the original. Details of the subject are hidden by the model. The model concentrates on the essence of the subject and to a given purpose. According to [BG01], a model should be able to answer questions in place of the subject. The answers provided by the model should be the same as those given by the subject itself.

Examples of models are easy to find in the domain of software engineering, as "everything is a model" [Bó4]. Let us reconsider the example of the domain of library management introduced

[2] For more definitions of the term "model" we refer to [MFB09] which summarizes the most popular definitions.

[3] For now, we will regard a model as an instance of a certain (model of a) language. The relationship between model and language is discussed in more detail in Sect. 2.2.4

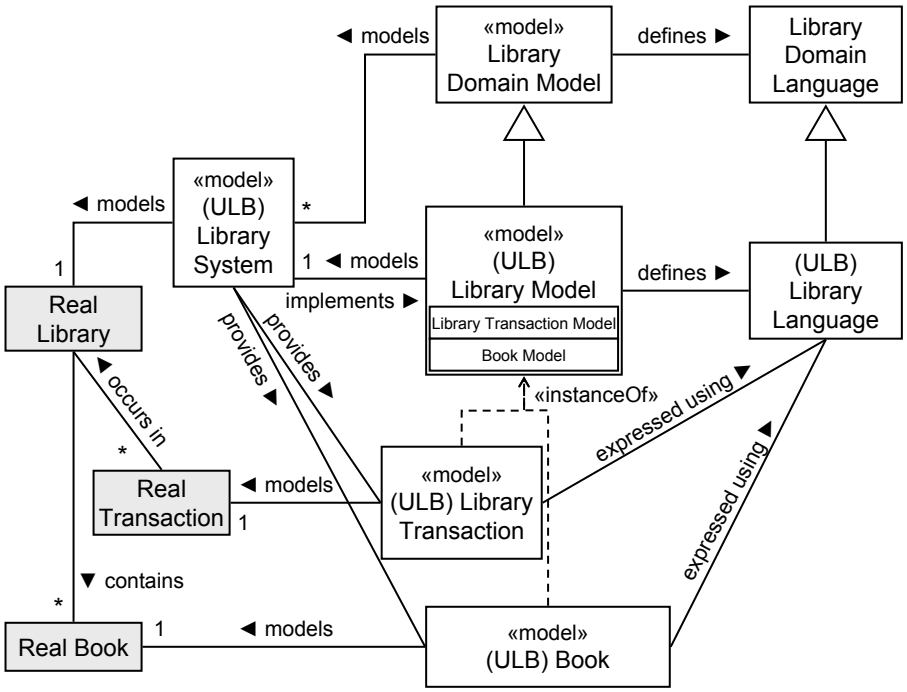


Figure 2.3.: Condensed view of models used in the library domain.

in Sect. 2.1. Figure 2.3 is a condensed view of the models participating in this domain. It depicts the subjects *Real Library*, *Real Book*, and *Real Transaction* that are modeled. In this example the *Universitäts- und Landesbibliothek Darmstadt* (ULB) is modeled (cf. Sect. 2.1). Furthermore, it depicts the involved model languages *Library Domain Language* $L_{LibraryDOM}$ and *Library Language* $L_{Library}$, as well as the occurring models[4]. Finally, the figure depicts the most important relationships between the depicted elements.

A library system $M_{LibrarySystem}$ models one specific real library. It might be thought of as a software system consisting of programs and databases that implement the system. The system is then used in the context of one real library. Note that $M_{LibrarySystem}$ is a model residing on the program level and is not the runtime instance of the program[5].

The content of a library, i.e., the real books available in a real libary, are represented as *models of the content of a library*. One physical instance of a book is represented by one model of a book M_{Book}. The transactions that occur in a real library, e.g., borrowing a book or querying the stock, are modeled by library transactions. One library transaction $M_{LibraryTransaction}$ models one real transaction. Consequently, $M_{LibrarySystem}$ provides library transactions and books. This has been explicitly depicted in Fig. 2.3 by the "provides" relationship. However, this relationship is also implicitly stated by the "models" relationship between $M_{LibrarySystem}$ and *Real Library* and the relationship between the real elements because it is part of the transitive closure of the latter relationships.

[4]Models are tagged with stereotype «model» in Fig. 2.3 and referred to as $M_{(abbrev.)NameOfModel}$.
[5]The reason will become clear in Sect. 2.4 when discussing modeling levels in the context of the MDA.

The domain model of the library domain $M_{LibraryDOM}$ (cf. Sect. 2.1) is a model, too. It is a model of many library systems and it defines a language $L_{LibraryDOM}$ that specifies the syntax and semantics of library models. One model of library models (i.e., $M_{LibraryModel}$ of the ULB) is a specialized model based on the domain model. Likewise to $M_{LibraryDOM}$, $M_{LibraryModel}$ also defines a language. But in contrast to the language $L_{LibraryDOM}$ defined by $M_{LibraryDOM}$, the language $L_{Library}$ defined by $M_{LibraryModel}$ is a special library language used in the context of one specific library system—the ULB.

The ULB library model contains, e.g., specific concepts and constraints that are applied in the real ULB but need not to be applied in other libraries, e.g., the Universitätsbibliothek Kassel (UBK). $M_{LibraryModel}$ serves as an implementation basis for one library system—the ULB library system. Therefore, $M_{LibraryModel}$ also has an implicit relationship to the real library. The specialization is depicted by a generalization arrow between $M_{LibraryDOM}$ and $M_{LibraryModel}$[6]. $M_{LibraryModel}$ aggregates the submodels $M_{LibraryTransactionModel}$ and $M_{BookModel}$. That is, these submodels are part of $M_{LibraryModel}$. A book model is an instance of the submodel $M_{BookModel}$ which is depicted by the «instanceOf» relationship between M_{Book} and $M_{LibraryModel}$[7]. $M_{BookModel}$ is depicted in more detail in Fig 2.4. Library transactions are modeled by the submodel $M_{LibraryTransactionModel}$. Both types of models, library transactions and books, are expressed using the library language defined by $M_{LibraryModel}$.

2.2.2. Abstract and Concrete Syntax

A model is represented either in *abstract syntax* or in *concrete syntax*. Figure 2.4 depicts concrete syntax representations of the models M_{Book} and $M_{BookModel}$ (which is a subset of $M_{LibraryModel}$) on the right-hand side and their abstract syntax representations on the left-hand side. The models in the lower part are M_{Book} models. The models in the upper part are $M_{BookModel}$ models. In this example, a book is more generally called *Publication*. A publication has a title and a number of authors.

In this thesis we will mostly[8] use the UML 2.2 object diagram notation to depict a model in abstract syntax. This notation is described in the classes package of the UML 2.2 Superstructure [Obj09b]. An *object* is an instance of a class and is depicted as rectangle containing the identifying name of the object followed by a colon and the name of the object's classifier—which are all underlined. Values of attributes that belong to the object's classifier are depicted as slots in a separate section of an object below the object's identifier. A *link* is an instance of an association and is depicted as line, optionally with arrow heads, which connects objects. The classifiers of objects and links are defined in the language used to express the model (cf. Sect. 2.2.4). It is important to notice that the elements depicted in abstract syntax are neither UML object nor UML link in terms of instances of classifier *InstanceSpecification* of the UML Superstructure (cf. Sect.2.3.3). Instead, they are instances of the classifier identified by the name after the colon.

[6] It is important to notice that the "generalization" relation between $M_{LibraryDOM}$ and $M_{LibraryModel}$ abstracts from library models in terms of generalization but not in terms of classification (cf. Sect. 2.2.3).

[7] An «instanceOf» relation between a type and its instance also denotes that the type is a model that abstracts from its instances in terms of classification.

[8] From Sect. 2.6 on we will also use graph notations to depict abstract syntax.

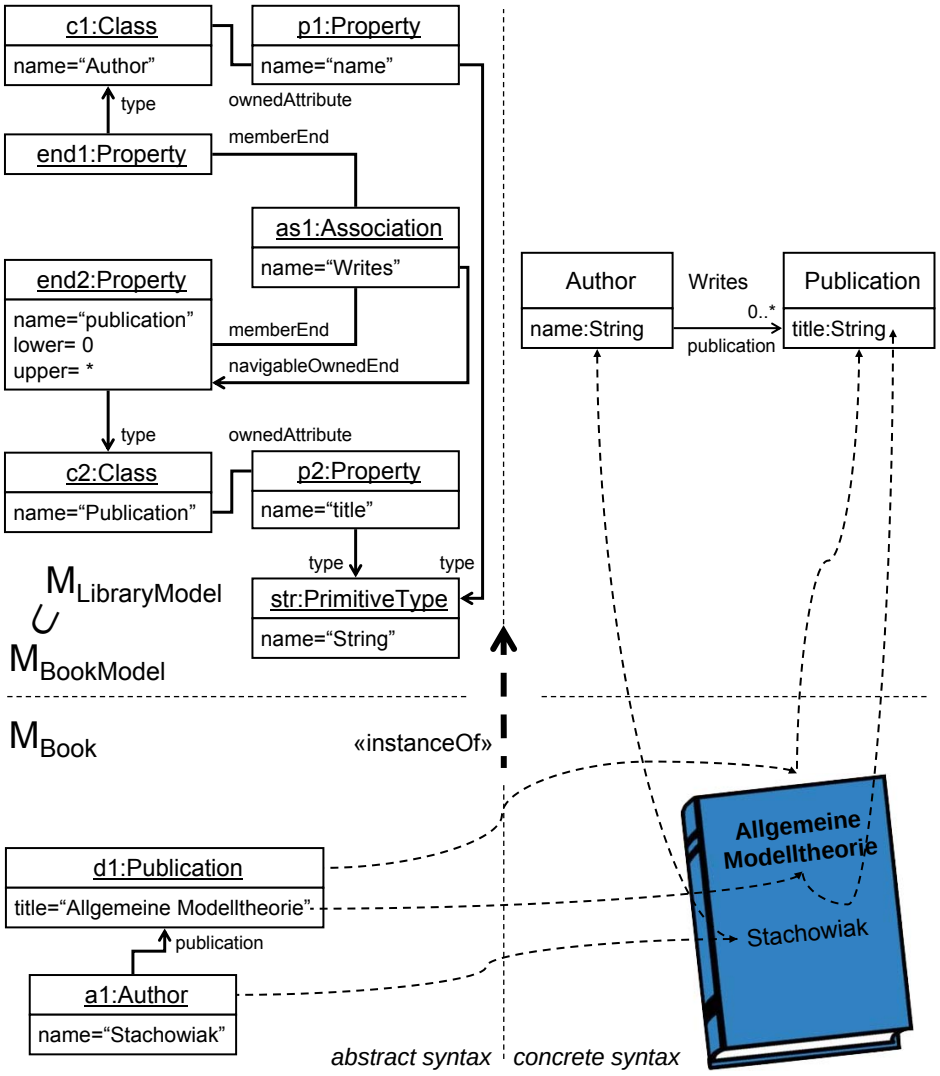


Figure 2.4.: Detailed view of models used in the library management domain.

The concrete syntax of a model depends on specific representations of the modeled subjects in the according domain. Such concrete syntax representations vary from printed text to graphical representations of the subjects. The concrete syntax of M_{Book}—as depicted on the lower part of the right-hand side—has been chosen to be as close to the real world book as possible[9]. The concrete syntax notation of $M_{BookModel}$—as depicted on the upper part of the right-hand side—is the common notation of class diagrams specified in the UML 2.2 Superstructure.

The left-hand side model $M_{BookModel}$ is an abstract syntax representation of its right-hand side counterpart. The model M_{Book}—on the lower left-hand side of the figure—consists of objects that are instances of *Publication* and *Author* and a link between both objects. The link is an instance of the association *Writes*.

The dashed arrows in Fig. 2.4 map elements of one model to corresponding elements. The horizontal arrows between the two representations of M_{Book} define a mapping of elements of the different representations of book models. We omitted the mapping from the class diagram representation of $M_{BookModel}$ to its abstract syntax representation due to visualization issues, but assume that the reader is familiar with the UML and is able to reproduce the mapping. The vertical arrows between M_{Book} and $M_{BookModel}$ define the «instanceOf» relation, between a model's elements and their classifiers.

2.2.3. Relationships Between Model and Subject

The library example shows that different kinds of models [Küh06b] occur while modeling a domain. The kinds of models are determined by the relation between two models or, more generally, a model and its subject. Figure 2.5 depicts the different roles of the models discussed in Fig. 2.3. The model of real world books M_{Book} is a so-called *token model* in the context of real books and library systems, hence the relationship to its subjects are called "token model of". Elements of a token model capture singular aspects of the original's elements. The abstraction process for creating a token model involves no further abstraction beyond projection and translation. So, a token model maps elements but does not classify elements by means of «instanceOf». In addition, $M_{LibrarySystem}$ and $M_{LibraryTransaction}$ are token models of a library and of transactions respectively, since they project a real library and a real transaction to the world of computers.

Contrary, a *type model* captures the universal aspects of subjects by means of classification, i.e., it contains elements that are classifiers of subjects. This is true for the model $M_{LibraryDOM}$ which classifies a number of library systems. Similarly, the model $M_{LibraryModel}$ is also a type model which classifies, e.g., book instances by providing the representatives *Author*, *Writes*, and *Publication* and defines kinds of library transactions, e.g., borrowing books. Hence, $M_{LibraryModel}$ has a "type model of" relation to M_{Book} and $M_{LibraryTransaction}$. It is also a type model of a library system, because it specifies the types and operations that are necessary to realize a library system. The last kind of model relation that occurs in our situation is the "super model of" relation between $M_{LibraryDOM}$ and $M_{LibraryModel}$. This relation maps

[9] This representation is also a model and no real world subject. A real world subject would be depicted by an exact copy of the book "Allgemeine Modelltheorie", e.g., a photography of the book (cf. Fig 2.7).

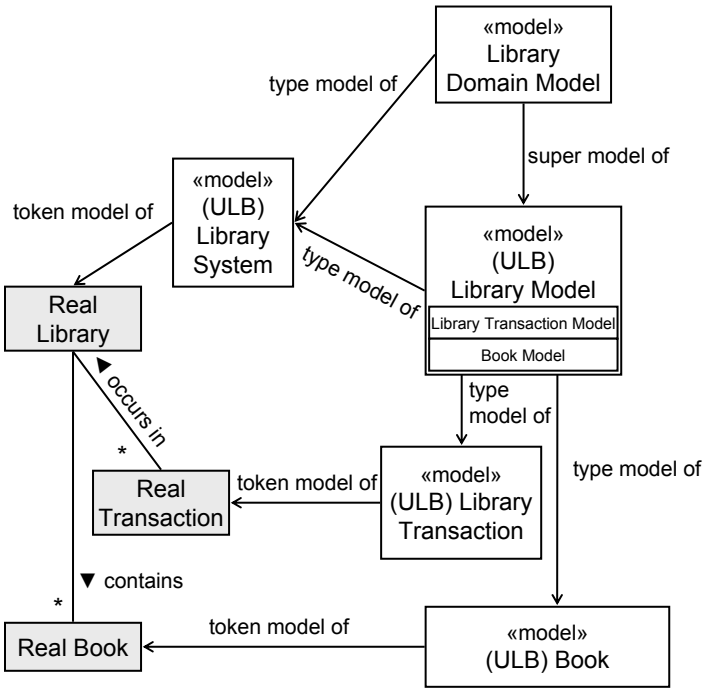


Figure 2.5.: Different kinds of models.

equivalent subject elements onto the same model element using an equivalence relation. For example, the specialized library models of the ULB and the UBK are mapped onto the library domain model using the generalization relation. It is important to notice that generalization only makes sense for type models and that generalization is not classification. Generalization maps *many concepts* to one (super-)concept, whereas classification maps *many elements* to one concept [Küh06b].

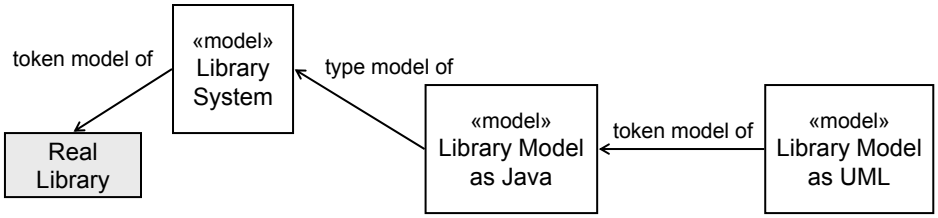


Figure 2.6.: Example of different roles a model may have.

Whether a model is a token model or a type model depends on the relationship to the modeled subject. Accordingly, a model may play the role of a token model in one context and the role of a type model in another context [Küh06b]. Let us consider the situation depicted in Fig. 2.6 that is an extension of Fig. 2.5. Now, $M_{LibraryModel}$ models a set of Java classes that implement the structure of a library system. In this context, $M_{LibraryModel}$ could be realized by UML diagrams that model Java classes. Thus, $M_{LibraryModel}$ does not classify the Java elements but only projects the Java language to the language used in $M_{LibraryModel}$.

Therefore, $M_{LibraryModel}$ has the role of a token model of Java classes in the scenario depicted in Fig. 2.6.

2.2.4. Languages

In model-driven engineering a model is written in a well-defined (modeling) language. Modeling languages have well-defined form (syntax) and meaning (semantics) that can be interpreted by a computer [KWB03]. Such formal modeling languages are often called *metamodel*. On the one hand, the term "metamodel" is accepted in the modeling community and widely used in practice. On the other hand, "metamodel" is not precisely defined, i.e., there is no consensus on what exactly a metamodel is or is not (cf. [Küh06b]). Therefore, we call the formal model that defines a modeling language a *language model* and discuss the relation of the term "metamodel" to the term "language model" in Sect. 2.2.5.

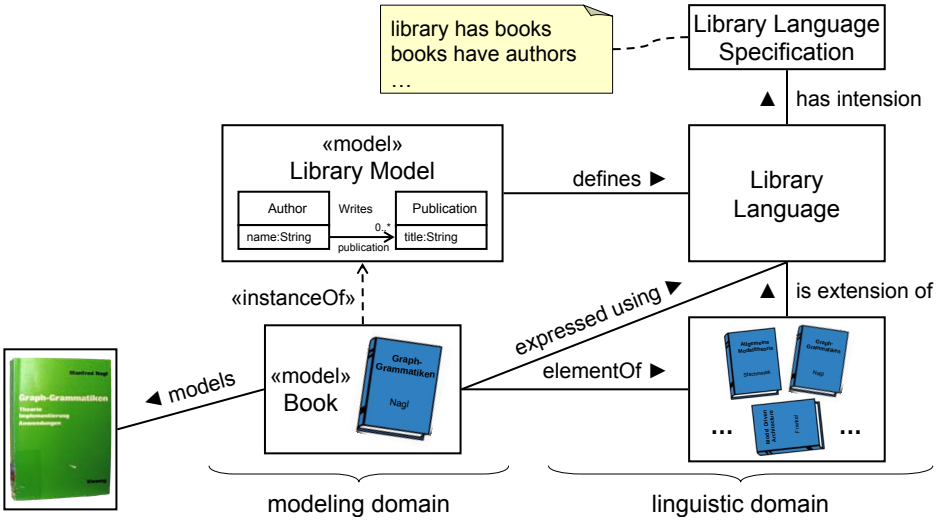


Figure 2.7.: Relations between models and languages by example.

Fig. 2.7 depicts the relationships between some of the models occurring in the library domain introduced in the previous sections and the library language. It depicts a model of a book that models a certain subject—a real book. The book model is expressed in a certain language—the library language. Each language has an intension which is stated by the specification of the language. The language has an extension which consists of all words or sentences which can be generated with the language. The book model is an instance of the model $M_{LibraryModel}$ because it is an element of the set of all sentences of the library language which is defined by $M_{LibraryModel}$. Henceforth we call models like $M_{LibraryModel}$ that define a language "language model" and use the following definition. A *language model* LM is a type model of models that defines a language L. A model expressed in this language is an instance of LM if it is an element of the set of all sentences which can be generated with the language L associated with the language model LM [Küh06b].

A model can be considered as a word, sentence, or instance of a certain language. A model is created, e.g., according to the rules of a formal grammar which may be part of a

language specification. The language defines the *syntax* and *semantics* of the models that are expressed in this language [Met05]. The syntax (also syntactic notation) is a possibly infinite set of elements that can be used in the communication, together with their meaning [HR00]. Given a model one can, e.g., check whether the model is syntactically correct regarding a certain language. Furthermore, the model must convey a certain unambiguous meaning in order to be useful. Therefore, carefully devised semantics assign unambiguous meaning to each syntactically allowed instance of the language [HR00]. Rigid well-formedness rules—also called *static semantics*—are required that prescribe the allowed form of a syntactically well-formed instance. In addition, rules are required that prescribe the *semantic definition* of the language—also called *dynamic semantics*.

> *A semantic definition for a language $\mathcal{L}$, or simply a semantics, consists of two parts: a* semantic domain*, which we denote generically by $\mathcal{S}_{\mathcal{L}}$, or simply $\mathcal{S}$ when there is no confusion, and a* semantic mapping *from the syntax to the semantic domain, denoted by $\mathcal{M}_{\mathcal{L}}$, or simply $\mathcal{M}$.*
>
> [Harel and Rumpe [HR00]]

The set of *syntactically correct* sentences of a specified language are (in our case) instances of MOF-compliant types. The set of *semantically correct* sentences is a subset of these sentences[10] from which sentences are removed that do not satisfy additional well-formedness rules, i.e., the static semantics. A model is *well-formed* or *valid*, if it is syntactically and semantically correct according to its language specification.

In our approach the syntax of a language is specified using the MOF (cf. Sect. 2.3.1). The constraint language OCL (cf. Sect. 2.3.2) is used to specify the static semantics. Well-formed models are, e.g., produced by applying production rules (e.g., SDM patterns) that belong to a (grammar-based) language model. So, these rules have to be specified carefully in order to not allow production of invalid models[11]. Story driven modeling and triple graph grammars, which are both modeling languages based on graph grammars (cf. Sect. 2.8.3 and Chap. 4), can be used to specify the semantic mapping of the language's syntax to a semantic domain.

Figure 2.8, which is based on Fig. 8 and 9 of [Küh06b], gives a schematic view on the situation depicted in Fig. 2.7. In addition, it depicts that the language model is expressed in a language, too, a *metalanguage*. This metalanguage[12] is defined by a formal *metalanguage model* of which the language model is an instance of. Each model resides on a certain level, called *modeling level*. Models that have an «instanceOf» relationship reside on different modeling levels[13]. This leads to a language definition stack where the lower part of Fig. 2.8 is

[10] In more informal terms, a model is *semantically correct* if an expert examining the model will agree that the model makes sense and does not contain errors according to the specified syntax and (static and dynamic) semantics of the language model.

[11] However, sometimes it is convenient and common practice to produce models which do not fulfill their constraints temporarily but are repaired later on.

[12] Figure 2.8 does not relate metalanguage and language because it depicts an object model and a language $L1$ need not to be expressed using the same metalanguage $ML1$ that is used to express the language model $LM1$ that defines the language $L1$. Instead a metalanguage $ML2$ could be used to express language $L1$.

[13] This condition does not hold for level M_3 in the MDA framework (cf. Sect. 2.4).

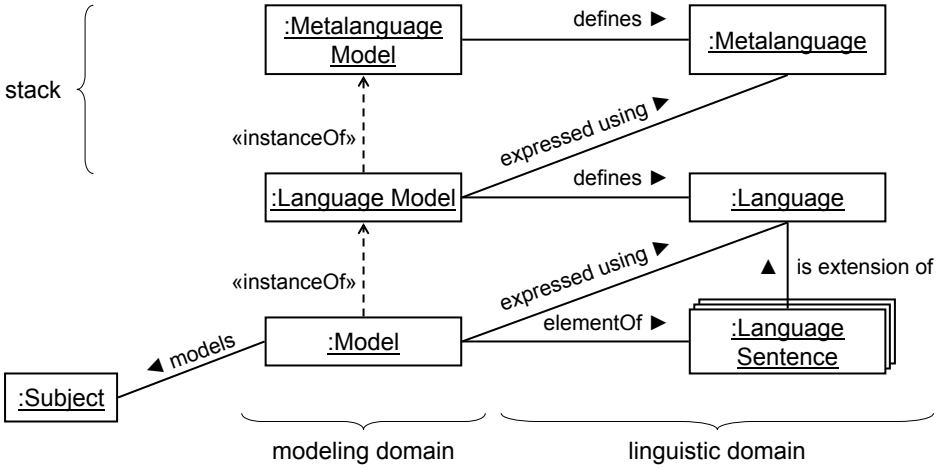


Figure 2.8.: Relations between models and languages—schematic with stack.

copied and shifted up one level. The "model"-role is assigned to "language model" and the role of the subject to "model"[14]. Such a language definition stack is a vital part of the modeling architecture presented in Sect. 2.4, which consists of a fixed number of modeling levels.

We have chosen to use the terms "language model" and "metalanguage model" as they seem to be rather intuitive. They are direct derivations of their counterparts in the linguistic domain. So, a language is defined by a language model and a metalanguage is defined by a metalanguage model.

For an in-depth discussion of the term "metamodel" one has to distinguish between linguistic and ontological instantiation (the interested reader is referred to [Küh06b]). In this thesis, we regard ontological language models (e.g., the author-publication model) as definitions of domain-specific languages (e.g., library language). This way we turn an ontological language model into a linguistic language model and, therefore, don't have to worry about linguistic and ontological instantiation. Keeping this at the back of the mind, we are now able to discuss the term "metamodel".

2.2.5. Metamodels

The prefix "meta" comes from the greek preposition and prefix *μετά*. In the context of the arrangement of the Aristotelian canon made by Andronicus of Rhodes, the preposition "meta" was used to describe a treatise about physics (cf. [CH11, Vol. 18: "Metaphysics"]). This treatise known as "metaphysics" was placed after another related treatise about physics from Aristotle in this canon. Therefore, *meta* was used as description for a treatise that was ordered spatiotemporal "after the physical treatises".

The modern meaning of "meta" is somewhat different and typically indicates a concept which abstracts from another concept. Therefore, the meaning of "meta" refers to something "above" (i.e., "about", "beyond", or "transcending") something. In addition, there is also a

[14] "Meta-Language Sentences" and "(Meta-)Language Specification" are intentionally not depicted in Fig. 2.8 to keep the figure more compact.

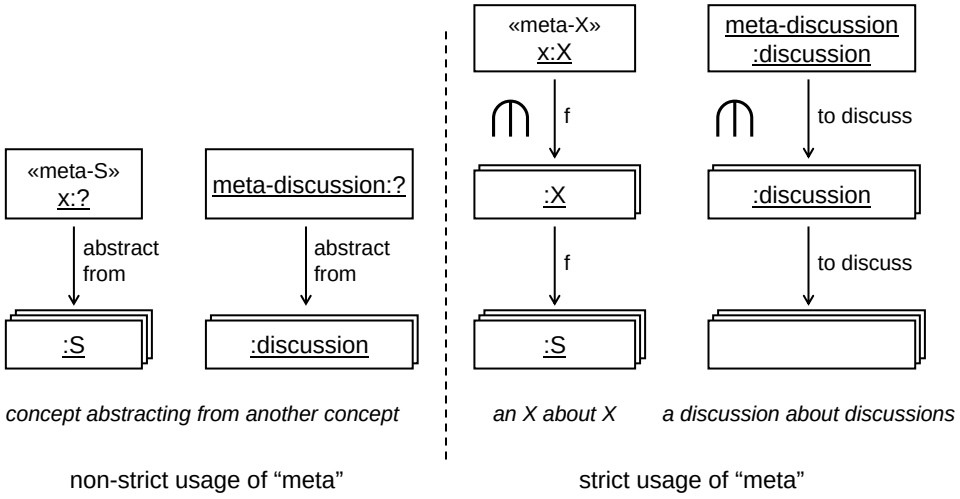


Figure 2.9.: Modern usages of the term "meta".

strict usage of "meta" that originated in the term "metatheorem" by Willard Van Orman Quine [Qui37]. Quine uses "metatheorem" to refer to a theorem about theorems which in turn are about systems. Douglas Hofstadter carried this strict usage of meta on, e.g., in the term "meta-description" which has the meaning of *descriptions of descriptions* [Hof99, chapter XIX "Artificial Intelligence: Prospects"]. In both cases, the prefix "meta" more strictly refers to "an X about X" (i.e., x about X | $x \in X$, X is a category, i.e., contains classifiers of elements). This indicates that an abstraction operation f is applied twice in order to achieve meta-ness. These two different modern usages of the term "meta" (non-strict and strict) are depicted in Fig. 2.9. Examples of these modern meanings of "meta" can be found in terms like "meta-discussion" which indicates a discussion about discussions[15], *"metadata" which relates to data about data*[16], *and "metalanguage" which is a language used to make statements about statements in other languages*[31]. Strict usages allow for self-reference, since if something (i.e., $x \in X$) is about the category to which it belongs, it can be about itself [Hof99, chapter XVI "Self-Ref and Self-Rep"]. Non-strict usage of "meta" indicates that x is about or beyond a system S but does not demand that x constitutes an S (i.e., it is permitted that $x \notin S$). It is also possible to use a non-strict "meta"-term x in a strict fashion and the other way round.

In the context of model-driven software engineering, a *metamodel* is typically thought of as a model that defines the syntax and static semantics of a language whereas the semantic mapping is often specified using a combination of formalisms and precise (but informal) natural language (cf. [Obj06] and [Obj09a]). Metamodels are often modeled using class diagrams in

[15] Funnily, this thesis is a publication about an extended version of the TGG approach which is also about other publications that are extended versions of the TGG approach (cf. discussion of other TGG approaches, which also discuss the TGG approach, in Chap. 8) and are also about the TGG approach. This makes this thesis a metapublication discussing publications (and the TGG approach) because (a) it is a publication and (b) it is about other publications which, for the purpose of comparison, discuss the TGG approach.

[16] " *noticed that it is hard to find examples of* meta-meta *and in common cases meta-meta is not necessary!*"[17] *like footnotes that transcend system barriers and provide information on information given in a system*

[31] cf. the metalanguage MOF introduced in Sect. 2.3.1

conjunction with textual constraint expressions. The relationship between a metamodel and the models that are expressed in the language defined by the metamodel is typically named «instanceOf». The models are denoted "instances" of the metamodel.

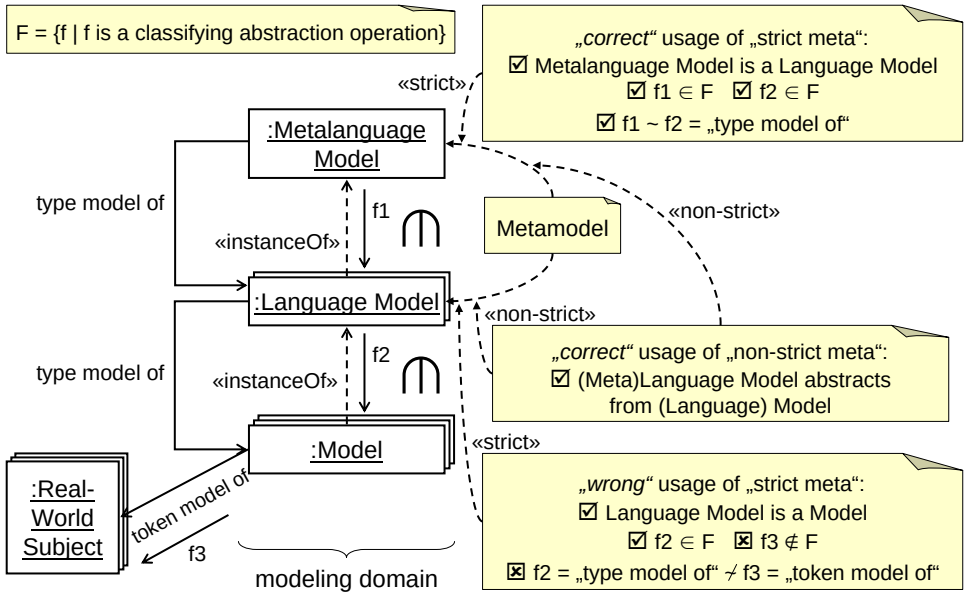


Figure 2.10.: Usage of the term "metamodel".

A main question in model-driven engineering is "when to grant the *metamodel status* to a model?". Kühne discusses this issue in [Küh06b] and proposes to differentiate between different kinds of models—"token model of" and "type model of"—regarding their relationship (i.e., representation, linguistic instantiation, ontological instantiation, and generalization). Summarizing this discussion, a token model of a token model is no metamodel by means of strict nor non-strict "meta" because a token model maps elements but does not classify elements by means of "instanceOf". Moreover, a type model of a type model is a metamodel because the relationship "type model of" is applied twice which results in meta-ness (in both strict and non-strict approaches).

Concluding, the term "metamodel" conveys different meanings if used strictly or non-strictly. Figure 2.10 depicts both the non-strict and the strict meaning of metamodel. In the non-strict approach, the term "metamodel" is used synonym to the terms "language model" and "metalanguage model" because a language model abstracts from its instances and, therefore, is "above" these models. Moreover, a metalanguage model is part of the language stack (cf. Fig. 2.8) which makes it "above" language models and models. The strict usage of "metamodel" allows to grant metamodel-status to metalanguage models because the operation "type model of" is applied twice in this context and a metalanguage model also *is a* language model. But, the strict usage disallows to grant metamodel-status to language models even though a language model *is a* model because the abstraction operation "type model of" is not applied twice. Instead, a different (non-classifying) abstraction operation "token model of" is applied in the chain of abstraction operations—between model and real-world subject.

Throughout this thesis we will use the non-strict definition of metamodel (cf. OMG's definition of "metamodel" in Sect. 2.3.1). This allows us to grant *metamodel status* to lots of models that are commonly thought of beeing metamodels: metalanguage models and language models[32]. Accordingly, we are able to reference language models and metalanguage models with the term "metamodel"—and, therefore, ensure "backwards compatibility". In addition, we are able to use the more precise term "metalanguage model" in order to reference a specific type of language models, i.e., language models that define a metalanguage. This leads to our definition of the term "*metalanguage model*": *a metalanguage model is a language model (that defines a metalanguage) that is a type model of language models that are type models of models.*

2.3. Modeling Languages

So far, we have discussed models and languages. Now it's time to present some formal languages that are applicable in the context of modeling software systems. We will focus on standardized modeling languages. First, we introduce the metalanguage MOF which is used to specify other modeling languages. Then we have a closer look at the constraint language OCL. Afterwards, we briefly explain parts of the general purpose modeling language UML that are relevant in this thesis. Finally, the idea of domain-specific languages and its relation to the UML is discussed. The relationships between these languages is depicted in Fig. 2.11 and will be explained throughout this section. We utilize version 2 of the languages MOF, OCL, and UML throughout this thesis. All three language specifications have been released by the Object Management Group (OMG), a computer industry standards consortium.

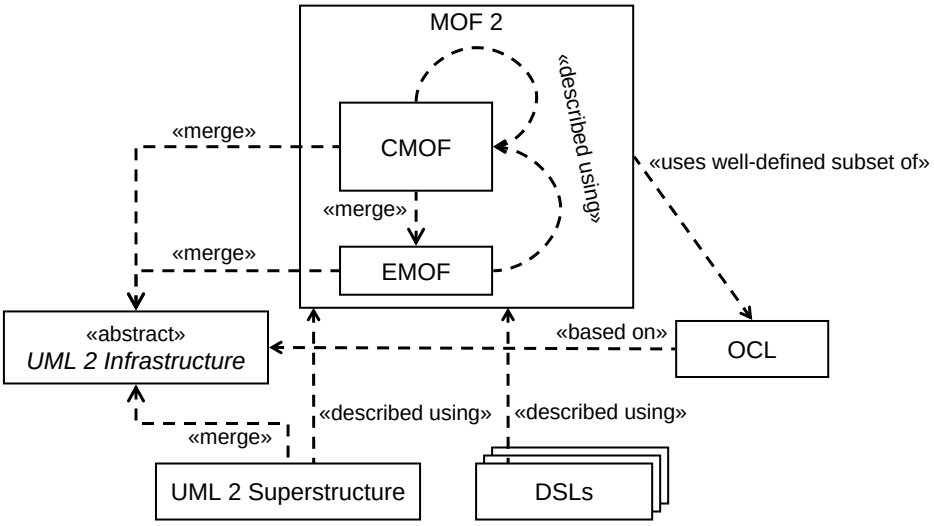


Figure 2.11.: Relationsship between selected modeling languages.

[32]Like in the MDA or in domain-specific modeling as we will see in Sect. 2.4.

2.3.1. MOF

The Meta Object Facility (MOF) [Obj06] is a modeling language that has "meta" capability. The MOF 2.0 specification states:

> *A metamodel is a model used to model modeling itself. The MOF 2 Model is used to model itself as well as other models and other metamodels (such as UML 2 and CWM 2 etc.).*

Therefore, MOF can be used to define and integrate a family of language models, i.e., languages that share certain elements, using simple class modeling concepts. In addition, domain-specific languages can be defined using the MOF. MOF provides concepts like classes, associations, properties, and modularization, as well as reflective capabilities, i.e., the ability to navigate from an instance to its classifier (its metaobject). The MOF specification uses a subset of UML, OCL (called *Essential OCL*, cf. [Obj10b]), and precise natural language to precisely describe the abstract syntax and semantics of the MOF.

The MOF specification has two compliance points. The Essential MOF (EMOF) which basically models simple classes with attributes and operations to fix the basic mapping from MOF to XML and Java. And the Complete MOF (CMOF) which is the language model used to specify other language models[33]—such as the UML. EMOF and CMOF are constructed by merging packages[34] of the UML Infrastructure. In addition, EMOF merges the MOF capabilities Reflection, Identifiers, and Extension. Whereas CMOF additionally merges EMOF. In the following we will have a closer look at CMOF and EMOF.

Chapter 8 "Language Formalism" of the MOF specification states:

> *In particular, EMOF and CMOF are both described using CMOF, [...]. EMOF is also completely described in EMOF by applying package import, and merge semantics from its CMOF description. As a result, EMOF and CMOF are described using themselves, and each is derived from, or reuses part of, the UML Infrastructure.*

Chapter 12 "The Essential MOF (EMOF) Model" of the MOF specification states:

> *A primary goal of EMOF is to allow simple metamodels to be defined using simple concepts while supporting extensions [...] for more sophisticated metamodeling using CMOF. [...]. EMOF, like all metamodels in the MOF 2 and UML 2 family, is described as a CMOF model. However, full support of EMOF requires it to be specified in itself, removing any package merge and redefinitions that may have been specified in the CMOF model. [...]. The reason for specifying the complete, merged EMOF model in this chapter is to provide a metamodel that can be used to bootstrap metamodel tools rooted in EMOF without requiring an implementation of CMOF and package merge semantics."*

[33] In this thesis we refer to CMOF when talking about MOF.

[34] "Package merging combines the features of the merged package with the merging package to define new integrated language capabilities. After package merge, classes in the merging package contain all the features of similarly named classes in the merged package." [Obj06]

Consequently, we may disregard the "complete, merged EMOF model" in our approach (because we rely on the CMOF) and use the "default" EMOF instead. Figure 2.11 depicts the relationships between "default" EMOF, CMOF, OCL, and the UML Infrastructure. In addition, it depicts the relationship of the UML Superstructure (which is also a metamodel in the UML family, cf. Sect. 2.3.3) and DSLs (cf. Sect. 2.3.4) to the MOF.

MOF defines some constraints on the language elements derived from the UML and defines some extensions. The most important differences to the UML are as follows. MOF adds an additional modeling element "Tag" to add key-value pairs to MOF elements. Tags are evaluated, e.g., by code generators. Moreover, the MOF constrains n-ary associations as defined in the UML to binary associations. In addition, the MOF provides concepts to query an instance of a language element during runtime about its language in a reflective way.

MOF and the derived technologies XMI and JMI are used for metadata-driven interchange and metadata manipulation, i.e., standardized exchange and manipulation of models (and metamodels). *XMI* (XML Metadata Interchange) [Obj07] is an OMG specification that defines a mapping from MOF to XML. *JMI* (Java Metadata Interface) specifies a mapping from MOF to Java—cf. Java Community Process JSR 040 [JSR02].

Revisiting Fig. 2.4 (cf. Sect. 2.2.2), the abstract syntax representation of M_{Book} and $M_{BookModel}$ indicate that the languages used to express the models provide certain classifiers. On the one hand, model M_{Book} indicates that $M_{BookModel}$ defines a language L_{Book} (which is part of the language $L_{Library}$) which provides the classifiers *Author*, *Publication*, and *Writes*. On the other hand $M_{BookModel}$ indicates that there must be another (meta-)language which provides the used classifiers. The classes *Class*, *Property*, *Association*, and *PrimitiveType* and some associations that relate these classes (e.g., an association between *Class* and *Property*) must be provided by this language. MOF provides these classifiers and, therefore, can be used to describe the library language $L_{Library}$.

2.3.2. OCL

The Object Constraint Language (OCL) [Obj10b] is a formal language which is used to describe expressions in UML and MOF based models. UML and MOF sometimes do not provide enough capability to express specific constraints about the elements in the model, such as complex invariants that must hold for the system being modeled. Therefore, it would be necessary to describe these constraints in natural language. OCL is used to describe additional constraints about the objects in the model instead of describing these constraints in natural language which often results in ambiguities. OCL is used to query models, to specify invariants on classifiers of a model, define pre- and post-conditions on operations and methods, and derive rules for attributes and associations. An OCL expression is guaranteed to be without side effects. So, a model is not changed when evaluating an OCL expression.

Basic types as boolean, numbers, and strings are supported by OCL. The types come along with basic operations like logic operations, $+, -, *, /$, and string operations. An OCL expression allows to navigate starting from a certain context element along the structure defined in a MOF or UML based language model.

Figure 2.12 depicts two OCL expressions that have been added to the library language model. An OCL expression is always in a certain context of an instance of a classifier. This

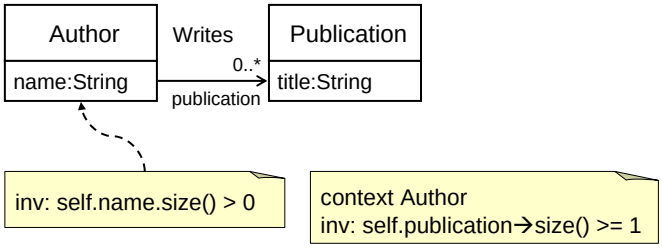


Figure 2.12.: Example of an OCL expression.

context is either established by the keyword "context" or—if depicted graphically—by a line that is connected to the context. To reference this instance from an expression the keyword "self" is used. Both expressions depicted in the figure are in the context of the class *Author* and are invariants that must hold for every instance of this class. The first expression demands that the size of the string containing the name of an author must be at least one. The second expression navigates from the author to its publications. The publications are returned as collection by the call to "self.publication". The collection operation "size()" returns the number of instances of publications attached to the author. Operations on collections are always preceded by an arrow following the name of the operation. Here, an author must have written at least one publication in order to be a valid author.

2.3.3. UML

The Unified Modeling Language (UML) [Obj09a, Obj09b] is a general purpose modeling language. In model-driven development it is used analogous to general purpose programming languages, like Java and C++.

The UML consists of the UML Infrastructure and the UML Superstructure. The Infrastructure constitutes the core of the UML and the MOF. The MOF 2.0 specification states:

> *The UML 2.0 Infrastructure Library uses fine grained packages to bootstrap the rest of UML 2.0. A design goal is to reuse this infrastructure in the definition of the MOF 2.0 Model.*

The Infrastructure contains elements and concepts required for modeling class-based structures. Therefore, it defines elements that are known in the context of class and package diagrams and concepts like modularization and visibility. The UML Superstructure provides a variety of diagram types used to model the structure and behavior of a system. Thus, it also specifies elements used to model class diagrams (similar but not equal to MOF class diagrams). In addition, it allows to model components, composite structures, deployments. The concepts supported for describing the behavior of a system include actions, activities, interactions, state machines, and use cases. UML models are both used for documentation purposes and for designing a system. But, in the context of model-driven engineering are also used as basis for generating (part of) an executable system from these models.

To show the difference between structural models based on the UML and structural models based on the MOF, Fig. 2.13 depicts a book model M_{Book} of the library system in both

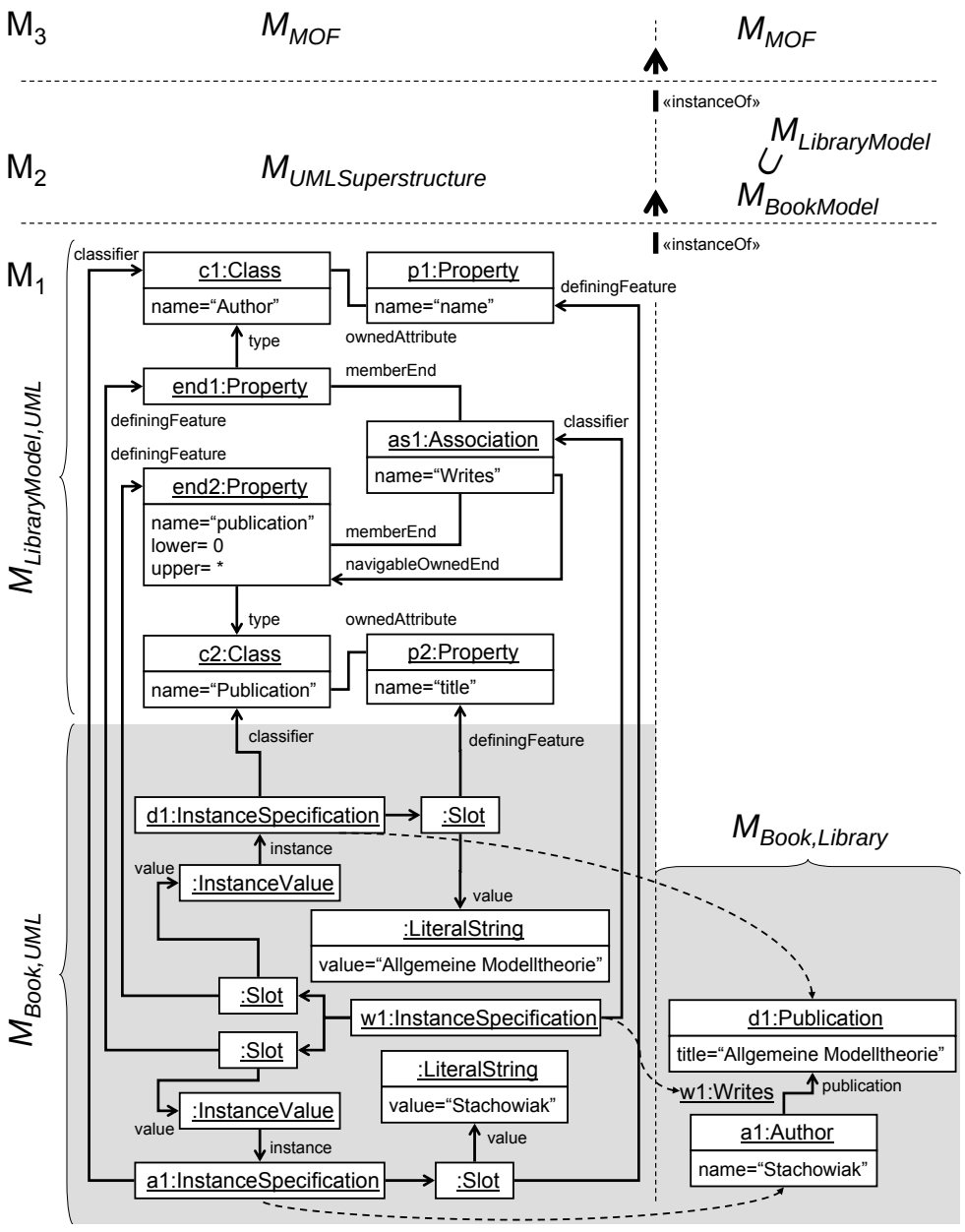


Figure 2.13.: Book model based on different languages: UML vs. DSL

approaches. Both versions refer to the same subject and look equal in concrete syntax (cf. lower right-hand side of Fig. 2.4). But, they are based on different languages. Therefore, both book models are depicted in abstract syntax in order to see the difference.

The model $M_{Book,Library}$ on the right-hand side is based on the domain-specific language $L_{Library}$ defined by $M_{LibraryModel}$[35] (cf. Fig. 2.4). On the contrary, the model $M_{Book,UML}$ on the left-hand side is based on the general purpose language UML Superstructure which is modeled by $M_{UMLSuperstructure}$. The models $M_{LibraryModel,UML}$ and $M_{LibraryModel}$ are almost identical[36], except that they are based on the—closely related—languages UML Superstructure and MOF respectively. Moreover, they are used in different modeling levels.

The publication object $d1$, the author object $a1$, and the "Writes" link $w1$, contained in the model $M_{Book,UML}$, are all instances of the type *InstanceSpecification*, being UML objects and UML links respectively which are defined in the UML Superstructure. An *instance specification* represents an instance in a system modeled with the UML. The title of the publication $d1$ and the name of the author $a1$ are so-called *slots*. Slots are linked to instances of type *ValueSpecification* (cf. [Obj09b, Figs. 7.6 and 7.8]). In this example, instances of *LiteralString* and *InstanceValue*, which are subtypes of *ValueSpecification*, are linked to the slots. Both slots related to the objects $d1$ and $a1$ are related to their defining feature which is the attribute *title* and *name* respectively. The according value is contained in an instance of the type *LiteralString*. The literal strings contain the publication title value *"Allgemeine Modelltheorie"* and the author name value *"Stachowiak"*. The link $w1$ has two slots that are associated with the member ends of the related *Writes* association $as1$. The values of these slots are *instance values*—instances of type *InstanceValue*—which are associated with the instance specifications of the objects $d1$ and $a1$ respectively. This way the ends of the link $w1$ are associated with objects $d1$ and $a1$ and, therefore, $w1$ links both objects.

The main difference of both M_{Book} models is the relation of elements to their language model. In $M_{Book,UML}$, the author and publication instances $a1$ and $d1$ are linked with their classifiers $c1$ and $c2$ defined in $M_{LibraryModel,UML}$. However, this is not an «instanceOf» relation to their classifying elements in the language model $M_{UMLSuperstructure}$ defining the UML language. An «instanceOf» relation would change the modeling level in the MDA framework (cf. Sect. 2.4). Each instance may only have one classifier on the next modeling level. Instead, the author and publication instances $a1$ and $d1$ are each linked to an object at the same level that represents their classifying element. Their classifier in terms of «instanceOf» is the element *InstanceSpecification* which indicates that $a1$ and $d1$ are both UML objects. Therefore, $M_{LibraryModel,UML}$ is no language model according to our definitions (cf. Sect. 2.2.4. Instead, it is a model, residing in the modeling level, which simulates a language model.

2.3.4. Domain-Specific Languages (DSLs)

A *domain-specific language* (DSL) is a language related to a particular problem domain. It is tailored to the activities in that problem domain. A DSL is applied by domain experts to

[35] more precisely $M_{LibraryModel,MOF}$ as it is based on the MOF language
[36] We omitted object "str:PrimitiveType" in $M_{LibraryModel,UML}$ to save some space.

solve tasks in the domain. It is created by language designers and domain experts. DSLs are only usable in the domain they have been designed for or in very closely related domains.

A contrary approach to DSLs is a general purpose modeling language (GPML) like the UML which is applicable in many domains. In general, it is easier for a domain expert to learn and comprehend a DSL that has been designed especially for his domain than a GPML. However, a software engineer who creates systems applied in different domains would reasonable learn one GPML rather than learning many DSLs. Hence, he would create the systems based on this GPML. To sum up, one has to reason whether the GPML or the DSL approach is advisable in a particular scenario before going for DSL or GPML.

As we have seen in Sect. 2.1, a domain(-specific) language may be the result of a domain analysis. The library language introduced in Sect. 2.2 is an example of a domain-specific language. In this thesis we use the metalanguage MOF to describe domain-specific languages. The resulting language models are called *DSL models*. Domain-specific models are then created based on these DSL models.

2.4. Model-Driven Engineering and the MDA

Likewise to high-level programming languages where competing approaches exist, there exist different approaches to model-based software engineering. The approaches have in common that they all use models as first-class artifact during the development process. That is, models are created and maintained and used throughout the development chain. Rather neutral terms when talking about model-based software engineering are *model-driven engineering (MDE)* and *model-driven development (MDD)*. The terms *Model Driven Architecture (MDA)* [KWB03] and *Domain-Specific Modeling (DSM)* [KT08] refer to specific approaches. Both approaches postulate to apply particular modeling and transformation languages, code generators, and activities. Which approach is suitable depends on the situation and even on the matter of taste. A software engineer who has to build an embedded system that has real-time requests would not apply standard Java but would instead use a more appropriate language in this situation. Likewise, this engineer would refuse inappropriate modeling approaches if he is asked to choose a modeling approach instead of a programming approach.

As our aim is to relate models based on different languages with each other the MDA framework proposed by the OMG fits our needs. The main ideas of MDA are stated in [KWB03] and [MKU04] as follows. Models are used as primary artifacts and are all written in a well-defined language. A platform independent model (PIM) describes a system without any knowledge of the final implementation platform. Whereas a platform specific model (PSM) describes a system having full knowledge of the final implementation platform. A PIM is developed based on a computational independent model (CIM). A CIM is a software independent model used to describe a business system.

PIMs are used as input for model transformations that refine the models to PSMs. Finally, code is produced from PSMs to realize an executable system for a particular platform. Examples of PIMs of metadata specified by the OMG include UML, MOF itself, CWM, SPEM, Java EJB, EDOC, EAI [Obj06]. Examples of PIM to PSM mappings include MOF-to-IDL mapping (defined in the MOF specification), MOF-to-XML DTD mapping (defined in the

XMI specification), MOF-to-XML Schema mapping (defined in the XMI production of XML Schema specification), and MOF-to-Java (defined in the JMI specification) [Obj06]. According to [KWB03], *automatic derivation of PIMs from a CIM is not possible, because the choices of what pieces of a CIM are to be supported by a software system are always human.* The OMG proposes to use the standard modeling languages UML, MOF, and OCL and in addition the model transformation language QVT [Obj08].

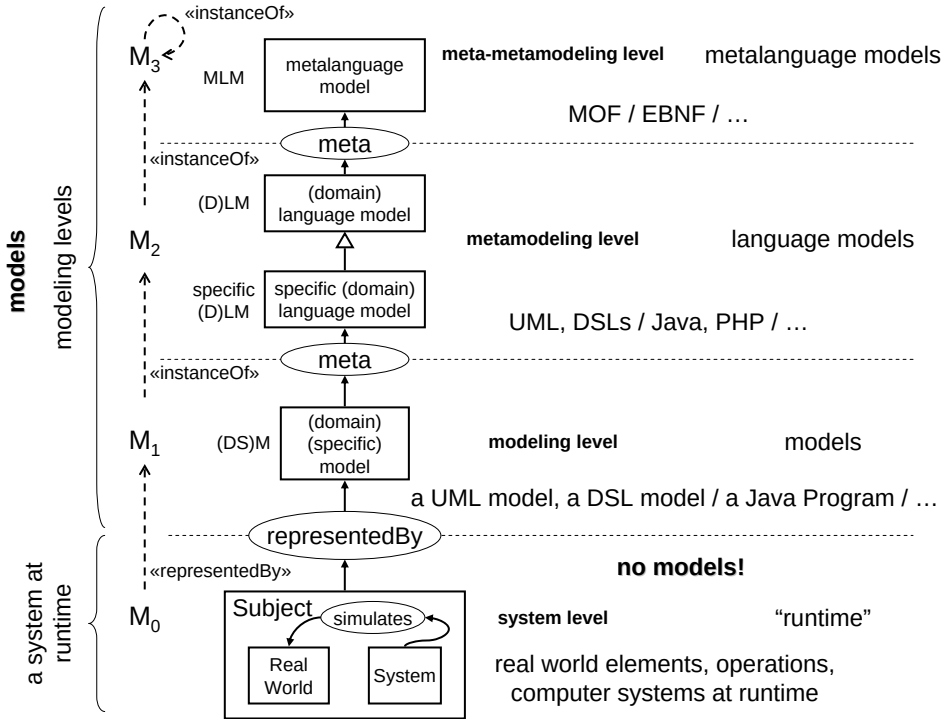


Figure 2.14.: Models and languages in the MDA 4-layer stack.

Figure 2.14—inspired by [KWB03, BG01, MKU04, Küh06b]—depicts the different modeling levels of OMG's 4-layer MDA framework. In this approach level M_0 depicts the runtime of a system simulating or representing real world elements, i.e., a program p that is executed. M_0 is not a modeling layer but a system, subject, or a situation being modeled. So, it does not contain models. Instead it contains runtime instances of real world elements and situations beeing modeled.

Levels M_3, M_2, and M_1 are the modeling layers. It is important to understand that these layers are not abstraction layers [BG01]. Therefore, modeling levels are changed by means of instantiation of classifiers, i.e., by «instanceOf» relations. Generalization does not change the modeling level. Another important constraint is that each model element has exactly one classifier in the MDA framework [BG01]. Changes between M_0 and M_1 are treated differently. There, the relation between the elements is not «instanceOf» but rather «representedBy» [BG01]. Going from M_1 to M_0 is similar to compiling a java program into bytecode and then executing the compiled program in a Java virtual machine. Going back to M_1 is achieved by the reflective mechanisms provided by the Java platform, i.e., a runtime object

may reflect about itself or others by querying language dependent information like classes, fields, method signatures, etc.

Level M_1 contains models. These may be object models (object diagrams) as well as type models (class diagrams), which depends on the language model present in the next level. We have discussed this situation in Sect. 2.3.3 where we explained the difference between models based on the UML Superstructure and models based on a DSL. If a program p is executed in level M_0 then M_1 contains a model of program p. Going meta we arrive at level M_2 that contains the language models of models in M_1. Level M_2 is the "metamodeling" level containing the modeling languages, i.e., the language in which p is written. When embedding the DSL approach into the MDA, domain language models (DLM), e.g., $M_{LibraryDOM}$, and specific DLMs like $M_{LibraryModel}$ (cf. Fig. 2.4) are located in this level. Finally, the "meta-metamodeling" level M_3 contains the metalanguage models, i.e., the language in which the language of M_2 is expressed. In our approach this is a model of the MOF language. The looping «instanceOf» relation indicates that a level at M_3 has the ability to describe itself in the MDA framework. A programming metalanguage equivalent to modeling metalanguages is the Extended Backus-Naur Form (EBNF). A popular metalanguage model is Ecore which is based on the EMOF and used in conjunction with the Eclipse Modeling Framework (EMF) in the software development environment Eclipse[37].

2.5. Model Transformation

The preceding section discussed the MDA framework and we learned that the MDA—as well as many other model-driven approaches—demand for model transformation. Subsequently, we will briefly present terms and features of model transformations used and developed in this thesis. For an in-depth discussion of model transformation approaches and a classification schema we refer to [CH03]. A taxonomy for model transformations is given in [MCG05]. Bidirectional model transformation approaches are discussed in [CFH+09] and [Ste10].

According to [KWB03], a model transformation takes models as input and produces other models as output. Therefore, a definition that describes how a model should be transformed is required. In [KWB03] such a definition is called *transformation definition* or *transformation specification* which consists of a collection of *transformation rules*. A transformation definition relates elements of the input language/domain with elements of the output language/domain to be independent of specific input models. Therefore, each transformation rule describes how and under which conditions one or more elements in the input language are transformed into one or more elements in the output language.

In common transformation terminology the input to a transformation is also called "*source*" whereas the output is called "*target*" because transformations are mostly executed in one direction only—from a source model to a target model. This is not sufficient when discussing *bidirectional* transformations which are executed in both directions, i.e., "*forward*" from source model to target model and "*backwards*" from target model to source model. That is, the source and target roles of a transformation are exchanged when transforming backwards. When using direction dependent terms this gets confusing when talking about backward transformations

[37] `http://www.eclipse.org`

because in this case the source of the transformation would be the target model. Usage of direction independent transformation terms avoids such collisions. Therefore, we will use the direction independent transformation terms "*input*" and "*output*" in this thesis whenever appropriate. Note that besides the model of the input domain a transformation may additionally operate on other models given to the translation process, e.g., an out-of-date model of the output domain in case of incremental transformations.

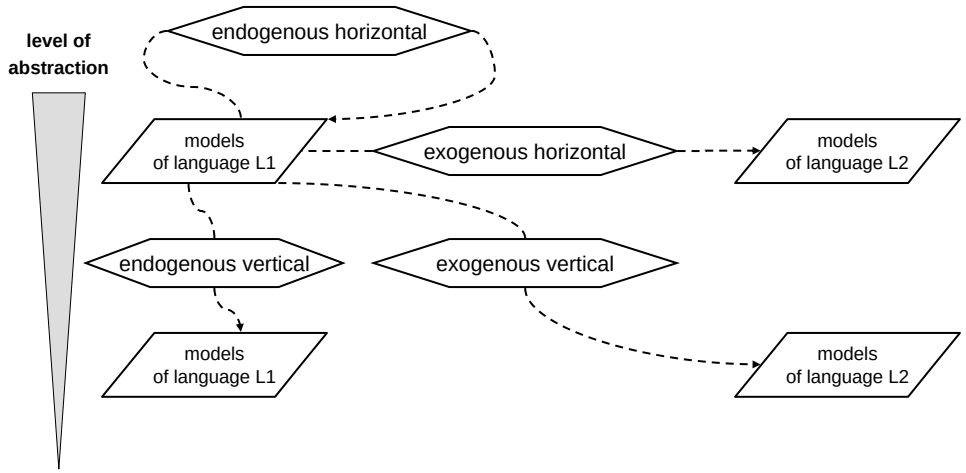


Figure 2.15.: Endogenous vs exogenous and horizontal vs vertical transformations.

Source and target language of models participating in a transformation may be equal or different (cf. Fig. 2.15). A transformation between models of the same language is called *endogenous transformation*, whereas it is called *exogenous transformation* if the languages are different [MCG05]. Exogenous transformations only make sense if the domains of the languages overlap, i.e., the languages share meaning or purpose (cf. Sect. 2.1). [MCG05] proposes to use the term "rephrasing" for an endogenous transformation, while to use the term "*translation*" for an exogenous transformation. A transformation where the abstraction level of input and output model are equal is called "*horizontal transformation*". Contrary, transformations that produce output models residing at a different abstraction level are called "*vertical transformation*" [MCG05].

According to [MCG05], endogenous/exogenous and horizontal/vertical are orthogonal dimensions of model transformations. Therefore, the terms can be combined. Examples of endogenous horizontal transformations are refactoring operations performed on a model and normalization according to some guidelines. Formal refinement is an example of endogenous vertical transformations. Language migration is an example of an exogenous horizontal transformation, assuming that the languages are located on the same level of abstraction, i.e., either M_2 or M_3. A transformation that generates code from a model for a specific programming language is an exogenous vertical transformation as the programming language is another language and resides on a lower level of abstraction than a modeling language.

Transformations preserve certain aspects of the input model in the transformed output model [MCG05]. The properties that are preserved differ on the type of transformation. For example, with refactorings the behavior of the modeled system needs to be preserved.

Exogenous transformations preserve, e.g., information and meaning between input and output models. However, some information might be lost due to an exogenous transformation depending on the capabilities of the output language to encode the informational features provided by the input language.

Likewise to models, transformation specifications are expressed in a particular language—a transformation language. Transformations are executed by a transformation tool. Which features are supported depends on the transformation language and on the transformation tool. Henceforth, we summarize the features discussed in [CH03] that are important in the context of this thesis. An "*in-place transformation*" reuses models given to the transformator when producing the output, i.e., produces output models by modifying the given models. Contrary, an "*out-place transformation*"[38] does not reuse given models. It creates new models leaving its inputs untouched, e.g., copies the input models and performs transformations on the copy. If a transformation rule is applicable at more than one match an application strategy can be deterministic, non-deterministic, or interactive (i.e., querying the user). Non-deterministic strategies include one-point application, where a rule is applied choosing one of the possible matches or by applying the rule concurrently to every possible match. Likewise, many rules could be applicable in one context. A transformation tool has to decide what to do in this case. A transformation language or tool may keep track of created, modified or deleted elements in the output models that resulted from the context of elements of the input models and the rule that was applied. Therefore, it can establish traceability links between related elements in source and target models.

Finally, we will briefly discuss bidirectional approaches to model transformation. We will go into detail in the subsequent chapters. According to [CFH+09], *bidirectional transformations are a mechanism for maintaining the consistency of two (or more) related sources of information.* Endogenous bidirectional out-place transformations, which result in unmodified input models and new output models, demand mechanisms for keeping these models consistent. The same holds for exogenous bidirectional transformations because such transformations always produce related source and target models that have to be synchronized if one of these models is modified later on. Therefore, bidirectional approaches to code generation and language migration (which are both exogenous) require mechanisms that enable roundtrip engineering in order to apply changes made in the model of the output domain back to the original model. Consequently, a bidirectional approach should feature traceability management between elements of source and target models.

2.6. Graphs

Graphs are well suited to describe the underlying structures of visual models. Especially transformations of visual models are naturally formulated by graph transformations [TEG+05]. Accordingly, graph-based transformation systems became of interest in the area of model-driven engineering in recent years [GGZ+05]. As the approach presented in this thesis is based on graphs we now introduce the notation and a formal definition of graphs. The definitions presented here base upon the definitions in [EEPT06].

[38] The term "out-place transformation" is not used in [CH03].

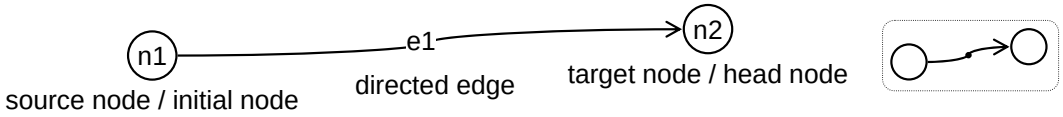


Figure 2.16.: Notation of graph elements: nodes connected by a directed edge.

Figure 2.16 depicts a *directed graph* which consists of two *nodes n1*, *n2* and one *directed edge e1* that connects the two nodes with each other. An *edge* starts at its *source node* and ends at its *target node*. The source node of *e1* is $s(e1) = n1$ and $t(e1) = n2$ is its target. Elements are optionally labeled[39]. A label is placed near its related element. An optional identifier of nodes is placed inside its graphical representation whereas an identifier of edges is placed near or on its graphical representation. Sometimes it is convenient to place a small black dot on an edge (cf. right-hand side of Fig. 2.16) when relating edges with other edges via morphisms.

The formal definition of graphs consisting of nodes and directed edges is as follows.

Definition 1. *Graphs.*
A quadruple $G := (V, E, s, t)$ *is a* graph *with elements*$(G) := V \cup E$*, where*
(1) V *is a finite set of* nodes *(also called* vertices*),*
(2) E *is a finite set of* edges*, and*
(3) $s, t : E \to V$ *are functions assigning sources and targets to edges.*

The *graph morphism* concept is a low level construct which plays a key role in the algebraic approach to graph grammars as we will see in the next section. Graph morphisms are used to relate two graphs by relating the nodes and edges of one graph with another graph, preserving source and target of each edge [Ehr79, EEPT06].

Definition 2. *Graph Morphisms.*
Let $G := (V, E, s, t), G' := (V', E', s', t')$ *be two graphs.*
A graph morphism $h : G \to G'$ *from* G *to* G' *consists of a pair of functions* $h := (h_V, h_E)$ *with* $h_V : V \to V'$ *and* $h_E : E \to E'$ *that preserve the source and target functions, i.e.,*
(1) $\forall e \in E$: $h_V(s(e)) = s'(h_E(e)) \wedge h_V(t(e)) = t'(h_E(e))$.
A graph morphism h *is* injective *(or* surjective*) if both functions* h_V, h_E *are injective (or surjective);* h *is called* isomorphic *if there exists a morphism* $h^{-1}G' \to G$ *from* G' *to* G *such that* $h \circ h^{-1} = id_{G'}$ *and* $h^{-1} \circ h = id_G$*. An isomorphism is* bijective*, which means both injective and surjective.*

A function is *injective* if every element of its codomain is mapped to by at most one element of its domain. A function *surjective* if (and only if) for every element y in the codomain there is at least one element x in its domain such that y is an image of x. Likewise to the vocabulary used for functions we say that G is the *domain* of h and G' is the *codomain*. Whenever we use the notation *graph morphism* (or *morphism* to simplify matters) we assume that the functions

[39] According to [EEPT06] labeled graphs and labeled graph morphisms can be considered as special cases of typed graphs and typed graph morphisms. So, we introduce typing later on but will not formally introduce labeling. However, the theory can also be applied to labeled graphs.

h_V and h_E are *total* functions, i.e., every element in V and E has an image in the according codomain. Contrary, we use the notation *partial graph morphism* if the functions h_V and h_E are partial functions, i.e., there could be elements in V and E which have no image in the codomain.

Definition 3. *Graph Operators.*
The binary operators $\subseteq, \cup, \cap, \setminus$ *are defined as usual. A* subgraph *A of B is denoted* $A \subseteq B$. *The* union *of two graphs with gluing of nodes and edges (with same identifiers) is denoted* $A \cup B$. *An* intersection *of graphs where the resulting graph contains only elements that are members of both intersected graphs, is denoted* $A \cap B$. *The* relative complement *where the resulting graph contains elements that are in A but elements with same identifiers in B are removed, is denoted* $A \backslash B$.
With $h : G \to G'$ *being a graph morphism,* $h(G) \subseteq G'$ *denotes that subgraph in* G' *which is the image of h.*

Note that a valid graph has no *dangling edges*, i.e., each edge must connect a source with a target node. The operators defined above delete such dangling edges in order to produce a valid graph. The gluing of graphs uses a technique called *pushout* which is described in the next section.

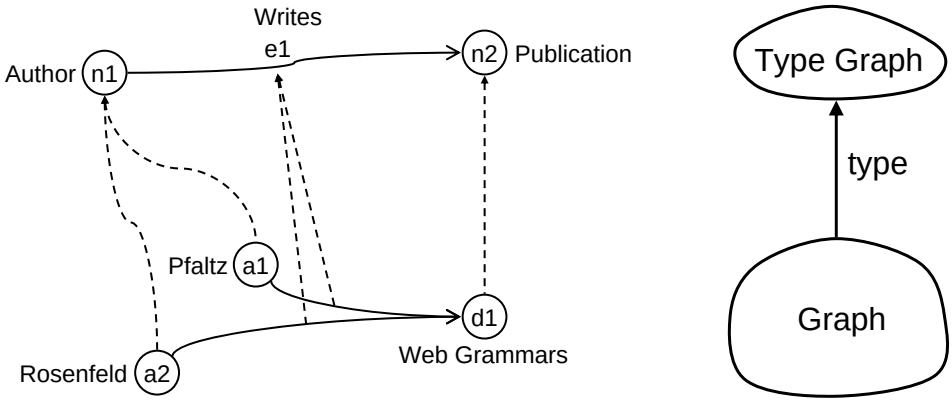
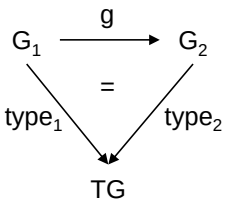


Figure 2.17.: Example of a typed graph.

Now, we extend the definition of graphs to typed graphs. This allows to distinguish between certain types of nodes and edges in a graph. Moreover, it allows to define graph schemata which are equivalent to metamodels in the world of models.

Definition 4. *Typed Graphs and Type Preserving Graph Morphisms.*
A type graph *is a distinguished graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$.
V_{TG} *and* E_{TG} *are called the node and the edge type alphabets respectively.*
A tuple $(G, type)$ *of a graph G together with a graph morphism* $type : G \to TG$ *is called* typed graph. *G is called* instance *of TG and TG is called* type *of G.*
A typed graph morphism $g : G_1^T \to G_2^T$, *with given typed graphs* $G_1^T = (G_1, type_1)$, $G_2^T = (G_2, type_2)$, *is a* type preserving graph morphism $g : G_1 \to G_2$ *iff the following diagram commutes:*



$\mathcal{L}(TG)$ *is the set of all graphs of type* TG.

Figure 2.17 depicts an example of a typed graph. The lower *graph* consists of two persons named *Pfaltz* and *Rosenfeld* which are identified by the nodes *a1* and *a2*. They have written a publication entitled *Web Grammars* which is identified by node *d1*. Thus, the nodes representing both persons are connected to *d1*. The *type graph*, also called *graph schema*, is depicted in the upper part of the figure. It consists of a node *n1* labeled *Author* and a node *n2* labeled *Publication*. An author is connected to a publication via edge *e1* labeled *Writes*. The *typed graph morphism* "type" (cf. right-hand side of Fig. 2.17) relates elements of the *instance graph* with elements of the *type graph*. The relation of pairs of elements is depicted by *morphism arrows* (cf. dashed arrows on the left-hand side of Fig. 2.17) that relate graph elements with elements of its type graph. In this example, the morphism relates Pfaltz and Rosenfeld with the author node and *Web Grammars* with the publication node. In addition, it relates both edges of the instance graph with edge *e1* in the type graph.

Definition 5 introduces constrained graphs. The regarded constraints are typed *graph constraints* (e.g., OCL invariants) in the spirit of [EEPT06], i.e, Boolean formulae over atomic typed graph constraints. A typed graph G fulfills a typed graph constraint c, e.g., if c is evaluated to *true*.

Definition 5. *Constrained Typed Graph.*
A type graph TG *with a set of* constraints $\mathcal{C}$ *defines a subset* $\mathcal{L}(TG,\mathcal{C}) \subseteq \mathcal{L}(TG)$ *of the set of all graphs of type* TG *that fulfill the given set of constraints* $\mathcal{C}$. *To simplify the following definitions, constraints are forbidden which are violated by the empty graph* $G_\emptyset$, *i.e., the empty graph* $G_\emptyset \in \mathcal{L}(TG,\mathcal{C})$. *Furthermore,* $\overline{\mathcal{L}}(TG,\mathcal{C}) := \mathcal{L}(TG) \backslash \mathcal{L}(TG,\mathcal{C})$ *denotes the set of all graphs of type* TG *that violate a constraint in* $\mathcal{C}$.

Now, that we have formally defined directed, typed, constrained graphs and relations between graphs we are ready to start discussing the rule-based transformation of graphs. In the following, we assume that all morphisms between graphs of the same type are type preserving.

2.7. Graph Grammars and Graph Transformation

Graph grammars originated in the late 1960's [PR69] and early 1970's [HJS71], were worked out in the subsequent years, and are still actively researched [CEM+06, SNZ08, EHRT08, ERRS10]. The motivation for their introduction was their application in the areas of pattern recognition, compiler construction, and data type specification [EEKR97]. But, graph grammars are also applicable in various other fields like, e.g., in software specification and development and they are relevant for industrial use [SNZ08].

Graph grammars are a natural generalization of formal language theory based on strings and the theory of term rewriting on trees. Hence, graph grammars are used to specify formal (graph) languages. Likewise, string grammars are used to specify formal (string) languages. The main component of a graph grammar is a set of *productions* that replace certain subgraphs in a *host graph* (or *input graph*) by another graph, i.e. transform graphs according to well defined rules. These productions specify the dynamic behavior of the graph language generated by their graph grammar. A production is, in general, a triple (M, D, E) where M and D are graphs (the mother and daughter graph respectively) and E is an embedding mechanism. Such a production can be applied to a host graph H whenever there is an occurrence of M in H. M is then removed from H and replaced by D using the embedding mechanism E [EEKR97, chapter 1]. There are different approaches to graph grammars and two main types of embedding: *gluing* and *connecting*. The so-called *algebraic approach* uses gluing, whereas the *algorithmic approach* (or *set theoretic approach*) uses connecting as embedding type[40]. In this contribution we focus on the algebraic approach, which uses pushouts as gluing mechanism, because the remainder of this thesis is based thereon. For an in-depth discussion of the algebraic approach and its mathematical background we refer to [EEPT06]. For a general overview, the history of graph grammars, and a discussion of other approaches and their relation to the algebraic approach we refer to [EEKR97].

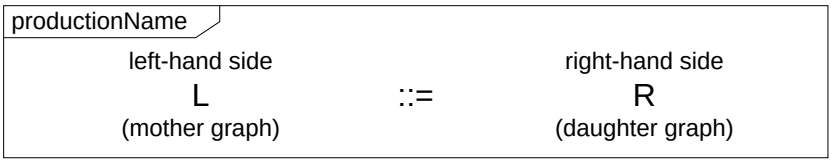


Figure 2.18.: Notation of graph production.

The main component of graph grammars are a set of productions. The basic idea of all graph transformation approaches is to perform direct derivations of graphs using productions $p : L \rightsquigarrow R$, i.e., to apply graph transformation rules [EEKR97, chapter 3]. The graphs L and R are called the left-hand side (*LHS*)—i.e., the mother graph—and the right-hand side (*RHS*)—i.e., the daughter graph —of the production respectively (cf. Fig. 2.18).

Figure 2.19 depicts a set of three productions named "createAuthor", "createPublication", and "addAuthor". Production "createAuthor" is an axiomatic production. It does not require any context as there are no elements on the left-hand side and produces a new author which is denoted by node *n1* labeled *:Author*. The preceding colon indicates that the name following the colon identifies the type of the node in the type graph. Production "createPublication" requires an author as context and connects this author to a new publication. Production "addAuthor" requires an author and a publication as context and connects both elements, indicating that the author also participated in writing the publication.

A *direct derivation* $G \overset{p@m}{\Longrightarrow} G'$ (also $G \overset{p@m}{\rightsquigarrow} G'$) produces a graph G' by fixing an occurrence of L by a *match* m in the given graph G and replacing the occurrence of L by R. To define a direct derivation step, pushouts are used which glue the graphs involved in the direct

[40] Recent algorithmic approaches (e.g., [Cam09]) additionally support gluing.

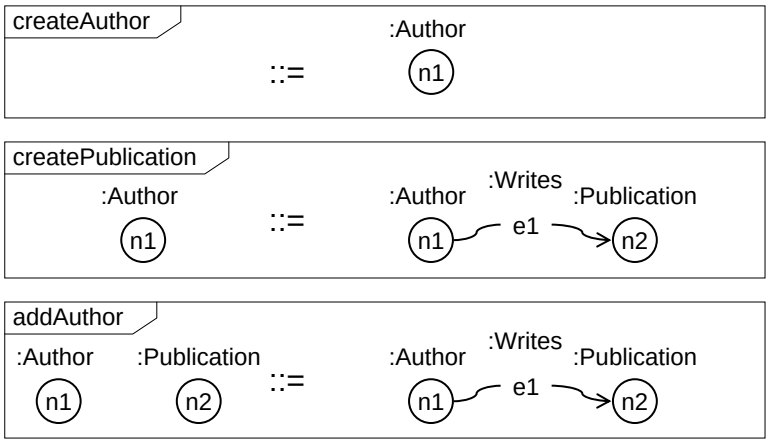
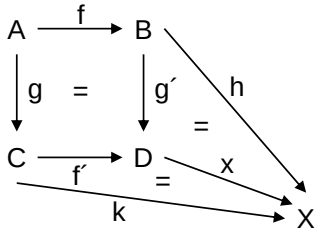


Figure 2.19.: Example of a production set.

derivation. There are two different approaches. The double-pushout (DPO) approach introduced in [EPS73], which is historically the first of the algebraic approaches, uses two gluing diagrams (i.e., pushouts) in the category[41] of graphs and total graph morphisms. The single-pushout (SPO) approach initiated by [Rao84] defines a derivation step as a single pushout in the category of graphs and partial graph morphisms.

The *pushout* construct (PO) is a technique to glue two graphs together along a common subgraph. The common subgraph is used as substructure and all other nodes and edges from both graphs are added [EEPT06].

Definition 6. *Pushout.*
Given morphisms $f : A \to B$ *and* $g : A \to C$ *in the category of graphs, a* pushout (D, f', g') *over* f *and* g *is defined by*
(1) a pushout object D *and*
(2) morphisms $f' : C \to D$ *and* $g' : B \to D$ *with* $f' \circ g = g' \circ f$
such that the following universal property is fulfilled:
For all objects X *and morphisms* $h : B \to X$ *and* $k : C \to X$ *with* $k \circ g = h \circ f$,
there is a unique morphism $x : D \to X$ *such that* $x \circ g' = h$ *and* $x \circ f' = k$*:*



We write $D = B +_A C$ for the pushout object D, where D is called *the gluing of* B *and* C *via* A *(over* f *and* g*)*. Alternatively, we write $D = B \cup C$ and demand that $A = B \cap C$.

[41] A *category* is a mathematical structure consisting of a class of objects, morphisms for each pair of objects, an associative composition operation $\circ$, and for each object an identity morphism *id*. In the category of graphs the objects are graphs from the class of all graphs.

An important fact is that the pushout object D is unique up to isomorphism (cf. [EEPT06, Fact 2.20]), i.e., all other objects X that are pushouts (X, h, k) via A over f and g are isomorph to D.

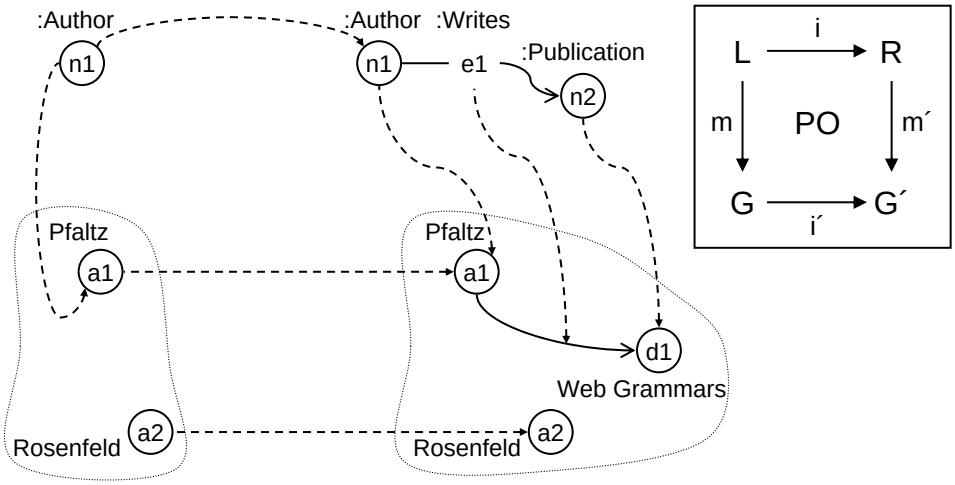


Figure 2.20.: Example of a pushout which glues two graphs G and R.

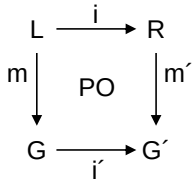
Figure 2.20 is an example of a pushout. Production "addPublication" (cf. Fig. 2.19) in the upper part of the pushout diagram is applied to G located in the lower left corner. G is a *predecessor graph* of the typed graph depicted in Fig. 2.17, where Pfaltz and Rosenfeld have not yet written the publication *Web Grammars*. In this example, the left-hand side of the production is the common subgraph. G and R are the graphs to be glued. The *derived graph* G' is created by adding an isomorphic copy of new elements in R, i.e., $R \backslash L$, to G and finally setting source and target nodes of new edges (here *e1*) to the elements in the copy of G identified by m and i (here the source node of *e1* is set to Pfaltz).

As we have seen in this example a direct derivation can be performed using one pushout in the category of graphs and total graph morphisms. This is only possible because all productions in the example production set (cf. Fig. 2.19) are *monotonic*[42], i.e., they do not delete any graph elements. But, this is not true in the general case where L also contains elements not present in R which are to be deleted (cf. Sect. 2.8.3). Both, the DPO and SPO approach support such non-monotonic productions. Monotonic productions are suitable for the extended triple graph grammar approach developed in this thesis. Hence, we formally introduce graph transformation based on monotonic productions only—instead of non-monotonic productions—and use these definitions as sufficient base for the extensions of the triple graph grammar formalism (cf. Chap. 4). For a formal definition of graph transformations based on non-monotonic productions we refer to [EEPT06].

Definition 7. *Monotonic Graph Productions.*
A typed monotonic graph production $p = (L \xrightarrow{i} R)$ *and* $L \subseteq R$ *consists of typed graphs* L *and* R*, called the left-hand side and the right-hand side respectively, and one injective typed graph morphism* i*.*

[42] In a monotonic production the left-hand side graph L is a subgraph of R.

Definition 8. *Graph Transformation with Monotonic Graph Productions.*
Given a typed monotonic graph production p, a typed graph $G \in \mathcal{L}(TG)$*, and a typed graph morphism* $m : L \to G$*, i.e., a match of* L *in* G*, a* direct typed graph transformation $G \overset{p@m}{\Longrightarrow} G'$ *(also* $G \overset{p@m}{\rightsquigarrow} G'$*) from* G *to a typed graph* $G' \in \mathcal{L}(TG)$ *is given by the following pushout diagram:*



A sequence $SEQ = (G_0 \Longrightarrow G_1 \Longrightarrow ... \Longrightarrow G_n)$ *of direct typed graph transformations is called a* typed graph transformation *and is denoted by* $G_0 \overset{*}{\Longrightarrow} G_n$ *or* $G_0 \overset{SEQ}{\Longrightarrow} G_n$.

Definition 9. *Graph Grammar and Graph Language.*
A graph grammar $GG = (TG, \mathcal{P})$ *over a graph type* TG *and a set of typed monotonic graph productions* $\mathcal{P}$ *generates the following language of graphs*
$\mathcal{L}(GG) = \{G | \exists$ *typed graph transformation* $\emptyset \overset{*}{\Longrightarrow} G\}$

According to Definition 9, a graph is in the language produced by a graph grammar if there exists a sequence of direct transformations that produce this graph. Let us validate whether the graph G_{PRWeb} at the bottom of Fig. 2.17 is in the language generated by the graph grammar $GG_{Bib} = (TG_{Bib}, \mathcal{P}_{Bib})$. The type graph TG_{Bib} depicted in the upper part of Fig. 2.17 consists of authors that are associated with their publications. The set of productions $\mathcal{P}_{Bib} = \{createAuthor, createPublication, addAuthor\}$ is depicted in Fig. 2.19.

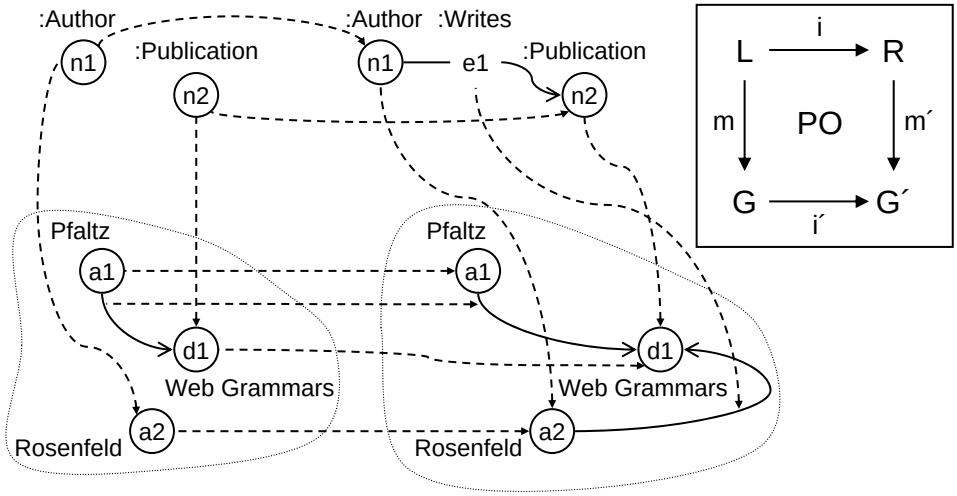


Figure 2.21.: Direct transformation with p=*addAuthor* that produces Fig. 2.17.

We must find a sequence of direct transformations that produce G_{PRWeb} starting with an empty graph. First, we produce the two authors by applying production "createAuthor" two times. Then, we apply production "createPublication". This leads to the situation depicted in

Fig. 2.20, where Rosenfeld is not yet connected to publication *Web Grammars*. Now, we are able to produce G_{PRWeb} by applying production "addAuthor" (cf. Fig. 2.21). Consequently, $G_{PRWeb} \in \mathcal{L}(GG_{Bib})$ because we have found a sequence of direct transformations.

These basic definitions of graphs, graph constraints, graph grammars, and graph transformations will serve as base for the triple graph grammar approach (cf. Chap. 4) which is used to specify bidirectional transformations in order to realize bidirectional language translation.

In order to avoid problematic situations that might generally occur in (non-monotonic) productions (i.e., potentially deleting productions) in the DPO approach, the match of the left-hand side in the host graph must satisfy an application condition, called the *gluing condition*. Otherwise, a DPO production is not applicable. The gluing condition consists of two parts: (a) the *dangling condition* which demands that if the production specifies the deletion of a node in the host graph then it must specify also the deletion of all edges in the host graph incident to this node; and (b) the *identification condition* which requires that every element of the host graph that should be deleted by the application of the production has only one pre-image in L, i.e., the matching morphism in the host graph is injective for to-be-deleted elements. Note that the triple graph grammar approach is based on the DPO approach. But, it is a special case with only one pushout in each domain and non-monotonic productions (i.e., productions that do not delete). Therefore, the *gluing condition* will never stop the application of a TGG production. Nevertheless, we discussed this condition here because of the *dangling condition* which inspired the *dangling edge condition* for triple graph grammars (cf. Sect. 5.3) which is a main result of this thesis.

2.8. Model Transformation Based on Graph Transformation

So far, we have investigated models, modeling languages, and model transformations on the one hand and graphs, graph schemata, and graph transformations on the other hand. Now, we will show the relation between the terms and concepts of the modeling domain and the domain of graphs. As both domains are closely related, some terms from both domains can be used interchangeably. Table 2.1 maps the terms that correspond to each other. The terms "attribute" and "label" in the domain of graphs are set in parenthesis because we do not use attributed and labeled graph grammars in this thesis. Instead, we simulate attributes (cf. Fig. 2.23) and labels (cf. Sects. 2.6 and 5.1). In Sect. 2.8.1 we present a possible solution for mapping models to graphs. This mapping definition will be used throughout this thesis to relate models to the world of graphs—especially to the triple graph grammar approach (cf. Chap. 4). Note that the presented mapping is just one of many possible solutions for mapping models to graphs (cf. discussion in Sect. 3.3.5). Section 2.8.2 then gives two examples of mapping first a model based on the DSL approach and second a model based on the UML approach to the world of graphs. Finally, Sect. 2.8.3 will complete this chapter by showing how model transformation can be realized by graph transformation.

domain of models	domain of graphs
language model / metamodel	graph schema / type graph
classifier	type
class	node type
association	edge type
property / attribute	(attribute)
model	graph
object	node
link	edge
slot with value	(label)
model transformation	graph transformation
transformation rule	production

Table 2.1.: Related terms of modeling domain and domain of graphs.

2.8.1. Mapping Models to Graphs

The next three figures depict the mappings from model elements to elements in the domain of graphs. Each model is mapped to a graph. The relation between an element in a modeling level M_i and its classifier in M_j is realized by morphisms in the domain of graphs between a graph G_i and its type graph G_j. Consequently, every graph that corresponds to a model of the MDA framework is related to a type graph. Morphisms are depicted as usual as dashed line with an arrow head pointing to the classifying element. So, morphism arrows depict «instanceOf» relations between an element (node or edge) and its classifier. If no morphism arrow is depicted in abstract syntax then the element in M_i has no image in its codomain M_j by means of one of the functions of the graph morphism $h : M_i \to M_j$ (cf. Def. 2). Note that every circle in abstract syntax depicts a node.

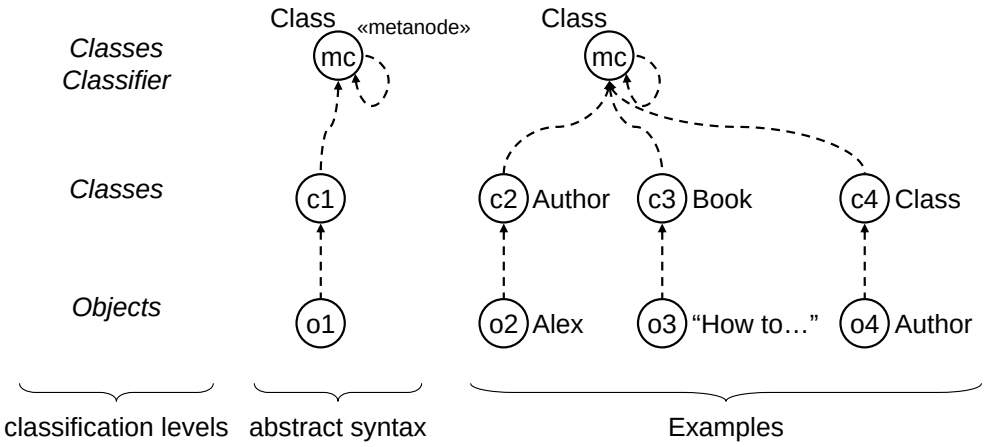


Figure 2.22.: Mapping of models to graphs—classes and objects.

Figure 2.22 depicts how classes and objects are mapped to graphs. The *metaclass mc* is a classifier of classes (*classes* are classifiers of *objects*). In addition, *mc* is an object which is an

instance of itself—which makes it a *metanode*. Therefore, the type (i.e., classifier) of *mc* is *mc*, too (self-reference like in MOF, cf. Sect. 2.4). This is depicted as a looping morphism arrow at *mc*. Consequently, *mc* is contained both in a graph and its type graph. Other instances of *mc*, located on the level of classes, are *c*1 to *c*4 which are objects representing a class, like the class of authors *c*2, the class of books *c*3, and the class of classes *c*4. Note that *c*4 is a semantic copy of *mc* which in normal circumstances is unnecessary. But, it helps to explain that even *mc*, as a metaclass, is an object. In addition, it shows that objects might be used on different levels of classification, like "Author" (*c*2 and *o*4) and "Class" (*mc* and *c*4). The level of objects contains instances of classes, like the author named "Alex", the book named "How to...", and the class named "Author".

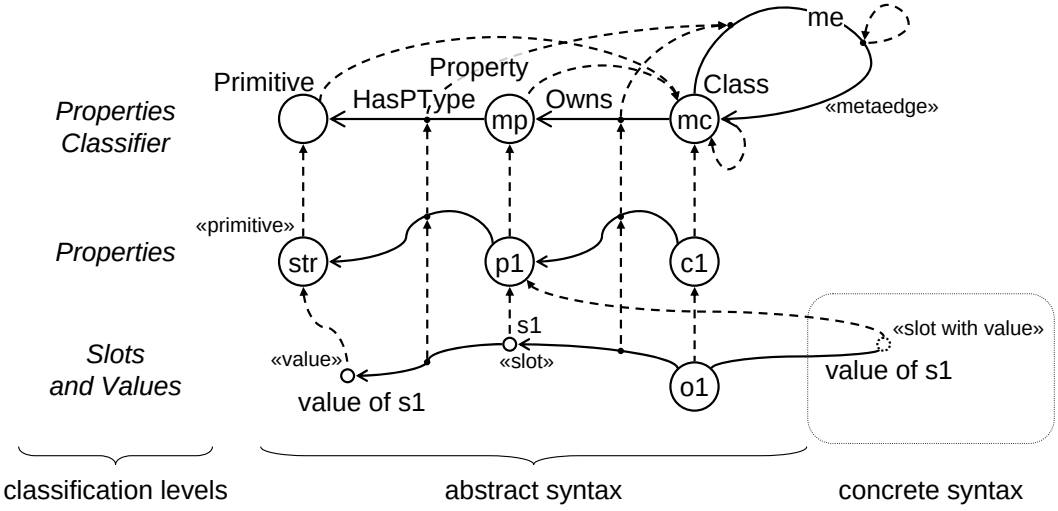


Figure 2.23.: Mapping of models to graphs—properties, slots, and values.

Figure 2.23 depicts how properties, slots, and values are mapped to graphs. The *metaproperty mp* is a classifier of properties (*properties* are classifiers of *slots*). It is connected with the metaclass *mc* which allows instances of *mc* (i.e., classes) to own properties—making them the *attributes* of a class. In addition, *mp* is connected with a classifier of primitive types. Both edges "HasPType" and "Owns" are related via morphism arrows to their classifying edge *me*. Edge *me* is an instance of itself—which makes it a *metaedge*. A *property* (e.g., *p*1) connected to an instance of mc (e.g., *c*1) and being of primitive type (e.g., *str* which is a classifier of strings) becomes a slot with value on the next instanceOf level, i.e., level "Slots and Values". A *slot* (e.g., *s*1) is a node which is an instance of a property (e.g., *p*1). A slot is connected to an object (e.g., *o*1) which is an instance of the class owning the property. The values (e.g., strings or numbers) are connected to the slot and the primitive values are depicted as label next to the value node. All edges depicted in Fig. 2.23 are related to their according classifying edge.

To be able to draw smaller graphs we introduce a concrete syntax representation of slots which is depicted on the lower right-hand side of Fig. 2.23. The dotted node with an morphism arrow to *p*1, marked as «slot with value», denotes a slot of *p*1. The context to its classifier *p*1 is made explicit by the morphism arrow.

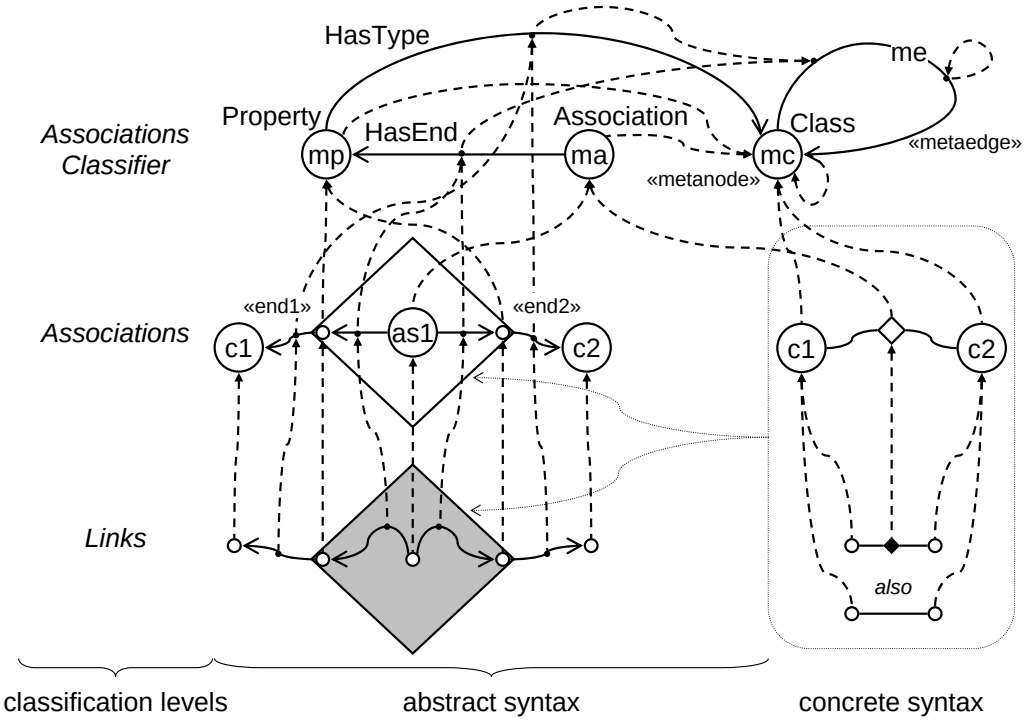


Figure 2.24.: Mapping of models to graphs—associations and links.

Figure 2.24 depicts how associations and links are mapped to graphs. The *metaassociation* *ma* is a classifier of associations (*associations* are classifiers of *links*). It is connected with the metaproperty *mp* via the edge "HasEnd" to realize the ends of an association. Similar to MOF, we fix the number of properties connected to an association to two. So, links are binary in this mapping approach. Therefore, an association has two ends which are instances of the metaproperty *mp*. The ends are optionally marked with «end1» and «end2». An end is connected to its association (e.g., $as1$) via an edge of type *HasEnd*. Moreover, a property connects to a class (e.g., $c1$ or $c2$) via an instance of edge *HasType*. A link consists of one node representing the link element and two edges that connect a link with its two slots that are instances of the classifying association's member ends. Moreover, each slot is connected with one of the linked objects. All edges depicted in Fig. 2.24 are related to their according classifying edge. The lower part on the right-hand side of Fig. 2.24 depicts a shorthand notation for associations and links. The white diamond[43] consists of an association connected with its two member ends. Each association consists of seven elements (1 association node + 2 end nodes + 4 edges). Links are represented as black diamonds in shorthand notation or as line between two objects without any arrow heads. Each link consists of seven elements (1 link node + 2 slot nodes + 4 edges). Note that lines belonging to an association or link have no arrow head in order to distinguish these lines from those lines representing an edge. Definition 10 gives a formal definition of links as a construct consisting of nodes and edges

[43] We decided to use a white diamond, in analogy to the white diamond that is used in the UML and in MOF, to represent an instance of the metaassociation *ma*.

based on the presented mapping.

Definition 10. *Let* link $l := (e_{t1}, s_1, e_{e1}, l_1, e_{e2}, s_2, e_{t2})$ *be a higher-level construct consisting of nodes and edges. Iff* o_1 *and* o_2 *are nodes representing objects,* s_1 *and* s_2 *are nodes representing slots, and* e_{t1}, e_{e1}, e_{e2}, *and* e_{t2} *are edges where*
$t(e_{t1}) = o_1$, $s(e_{t1}) = s_1$, $t(e_{e1}) = s_1$, $s(e_{e1}) = l_1$,
$s(e_{e2}) = l_1$, $t(e_{e2}) = s_2$, $s(e_{t2}) = s_2$, $t(e_{t2}) = o_2$,
then link l connects *the objects* o_1 *and* o_2. *Node* l_1 *is called the* link node *of link* l.

With these mappings we have constructed graphs that correspond to concepts from the modeling domain. Each graph is related with its type graph via a graph morphism $h_i : G_i \rightarrow G_j$ between graphs G_i and G_j with $h_i := (h_{V,i}, h_{E,i})$. These morphisms map each node of G_i to its classifying node in G_j with the function $h_{V,i}$ and each edge of G_i with its classifying edge in G_j with the function $h_{E,i}$. In addition, both functions $h_{V,i}$ and $h_{E,i}$ preserve source and target functions of edges (cf. Figs. 2.22, 2.23, and 2.24 and Def. 2). Consequently, we have constructed total graph morphisms because every node and edge in G_i have an image in their codomain G_j and the morphism properties are satisfied. Note that in general the morphisms are non-injective because a classifier is referred to by many instances. Accordingly, not every element of the codomain of functions $h_{V,i}$ and $h_{E,i}$ (i.e., a classifier) is mapped to by at most one element of its domain (i.e., an instance) in the general case. However, a morphism is surjective because for every classifying element (in the codomain of functions $h_{V,i}$ and $h_{E,i}$) there exists at least one instance (in the domain of functions $h_{V,i}$ and $h_{E,i}$).

2.8.2. Models Mapped to Graphs: Two Examples

Based on the mapping definitions given in the previous section, we now discuss the translation process of two models and their classifying models into corresponding graphs that are related via morphisms. The first of these models is based on the DSL approach, the second model on the UML approach. Producing graphs from these models is straightforward. The mapping definitions depicted in Figs. 2.22, 2.23, and 2.24 are applied to every object, slot, and link starting in the modeling level M_1. This produces a graph G_1, its type graph G_2, and its *metatype graph* G_3 which correspond to the models located in M_1, M_2, and M_3.

Figure 2.25 depicts a model of the library example on the left-hand side (cf. left-hand side of Fig. 2.4 that depicts the abstract syntax of a quite similar model). The library's type model is located in M_2 and is an instance of the MOF (DSL approach). Instances of the library type model are located in M_1. Graphs that correspond to the models are depicted on the right-hand side of Fig. 2.25. The graphs are depicted in concrete syntax to simplify matters.

First, objects, links, slots, and values are translated to nodes and edges and added to graph G_1 that corresponds to the model located in M_1. This produces a representation of two authors that have written a publication in the world of graphs. After all elements of M_1 have been translated, all not-yet-translated elements in M_2 and afterwards in M_3 are translated. This adds additional information to the type graphs G_2 and G_3, e.g., the name of association $as1$, i.e., *Writes*. Translating not-yet-translated objects of M_2 and M_3 (i.e., classifying objects in the (meta-)language model) will result in morphisms that are not surjective because the corresponding nodes in type graphs G_2 and G_3 have no instances in G_1 and G_2 respectively.

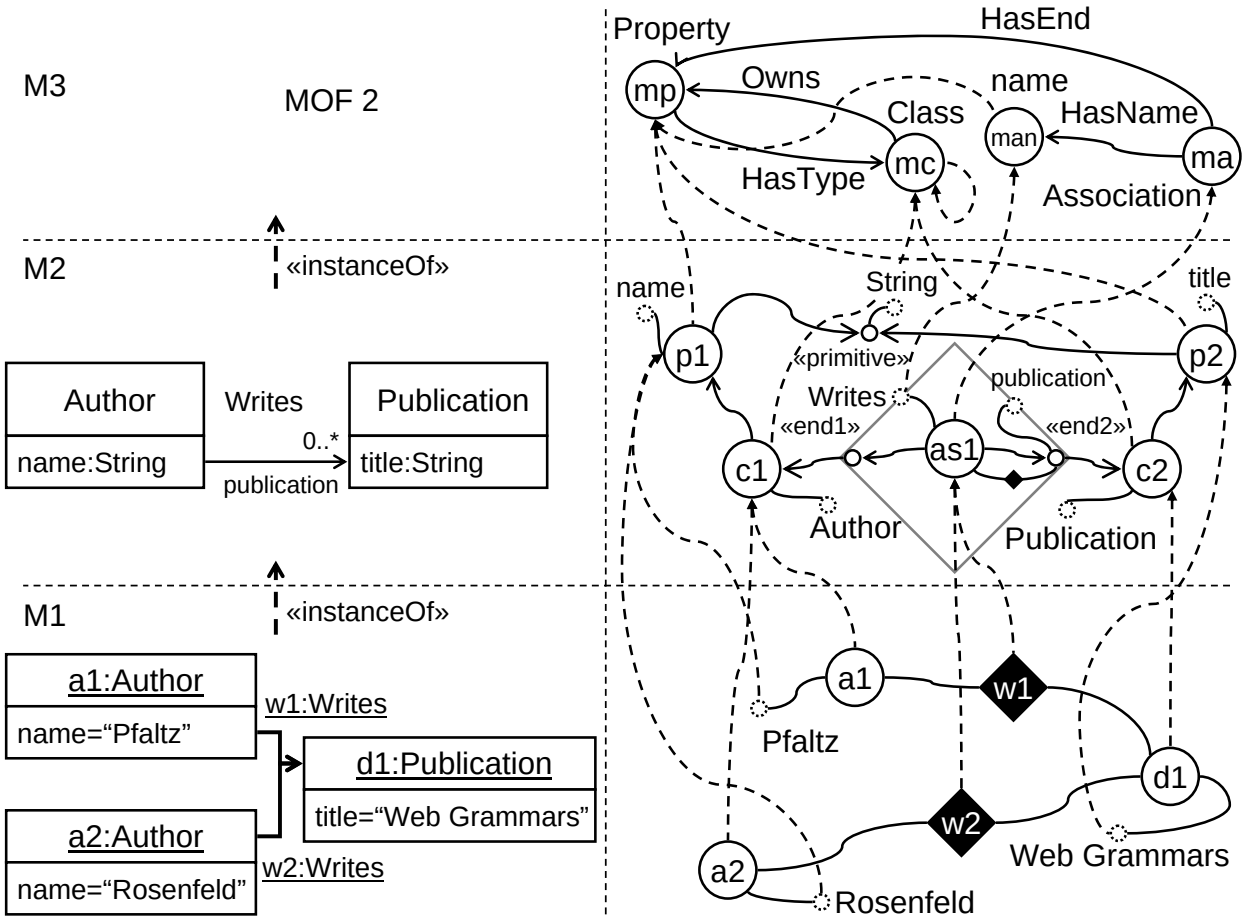


Figure 2.25.: Models mapped to graphs—in the context of a DSL.

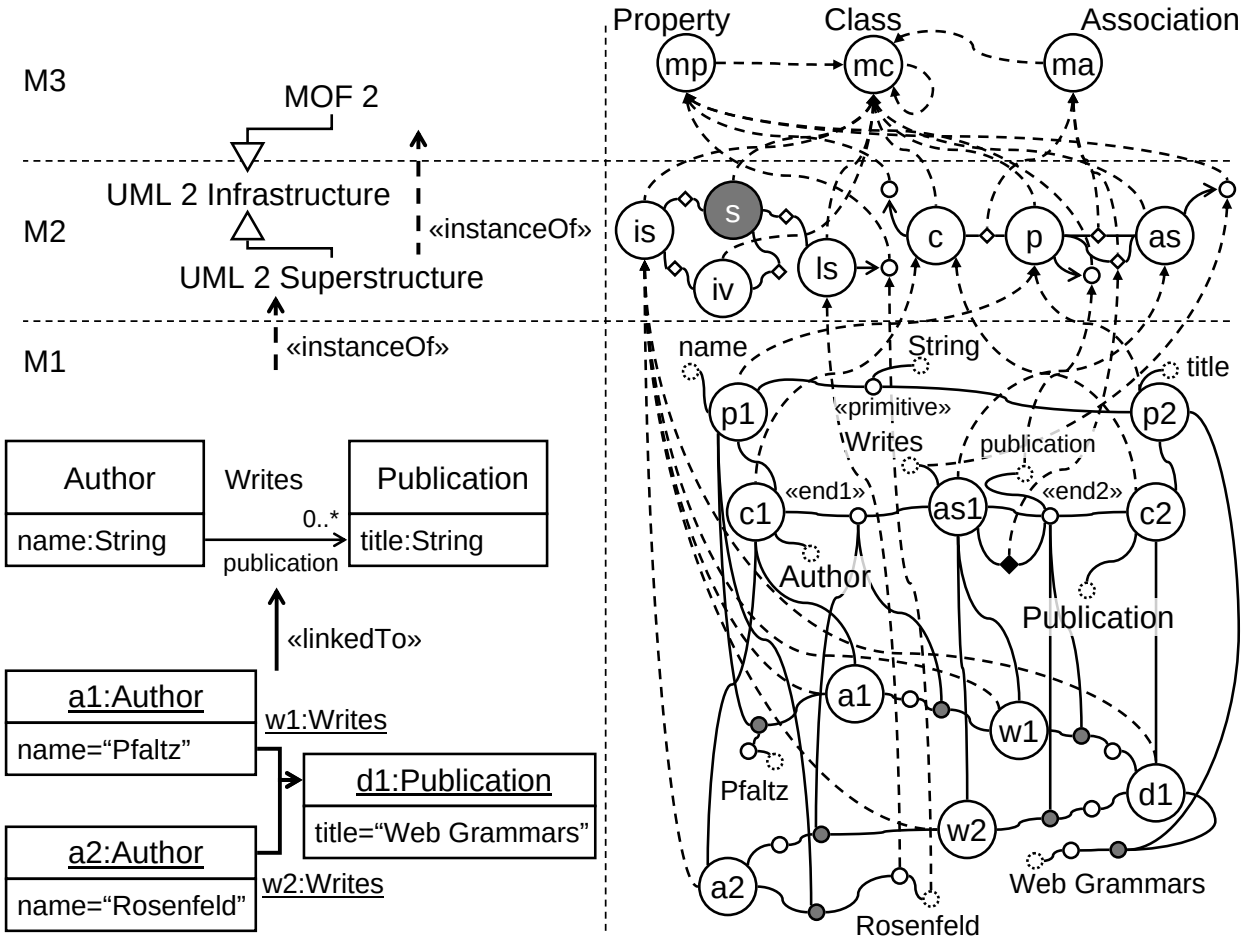


Figure 2.26.: Models mapped to graphs—in the context of the UML Superstructure.

Translating not-yet-translated slots, values and links of M_2 and M_3 will also result in surjective morphisms because these elements do not classify other elements and, therefore, have no instances. However, if the latter elements are not translated information will be lost, e.g., the name of association $as1$. When translating objects, links, slots, and values contained in M_2 and M_3, G_3 will be used as type graph and metatype graph of G_2 and G_3 respectively. That is, classifying elements like mc, ma, mp, and me contained in G_3 are reused.

The morphism arrows from link nodes $w1$ and $w2$ to association node $as1$ are a shorthand notation (cf. Fig. 2.24). The origin of these morphism arrows is the link node inside the black diamond. The link between association node $as1$ and its second member end, which is depicted as black diamond, corresponds to the link *navigableOwnedEnd* which is depicted on the left-hand side of Fig. 2.4 as link *navigableOwnedEnd* between $as1$ and $end2$.

Note that we omitted some elements and morphism arrows in G_2 and G_3 and depicted only the most important elements. Therefore, the classifying association of link *navigableOwnedEnd* is not depicted in G_3 (cf. [Obj09a, p. 111, "11.3 Classes Diagram"]). Moreover, we omitted the primitive type classifier *Primitive* and the metaedge me (cf. Fig. 2.23) which are both contained in G_3. According to the mapping definition depicted in Fig. 2.23, a new property man is created as an instance of type *Property* (i.e., the classifying node mp) when translating slot *name* of association *as1*. Property man (i.e., name of metaassociation ma) is connected to ma via a new edge *HasName*. The same applies to the names of authors, publications, and primitive types which would result in additional properties mcn, mpn, and $mprimn$ that are not depicted in Fig. 2.25. Likewise, G_2 does not depict the slots and values of the attributes *lower* and *upper* which are owned by metaproperty *Property* (cf. [Obj09a, p. 95, "10.2 Classes Diagram"] and property *end2* depicted in Fig. 2.4). However, the missing elements can be easily reconstructed by referring to the mapping definitions discussed in Sect. 2.8.1.

Due to our concrete syntax definitions of graphs, a model's representation as graph in concrete syntax is almost similar to the model's abstract syntax representation. This becomes clear if we compare the graphs depicted on the right-hand side of Fig. 2.25 with the abstract syntax representation of their corresponding models depicted on the left-hand side of Fig. 2.4.

The resulting graphs G_1, G_2, and G_3 look different if the library language model and library models are based on the general purpose modeling language specified in the UML Superstructure (UML approach). Both approaches (DSL approach and UML approach) use a MOF metalanguage model in M_3. But in the DSL approach, the library language model is directly based on the MOF, whereas in the UML approach, the library language model and its instances are both based on the UML Superstructure. This leads to different «instanceOf» relationships and links between objects. The graphs which correspond to models of the UML approach are depicted in Fig. 2.26. Note that all lines without arrow head that connect two objects are links—the black diamond has been omitted—and not edges (cf. Fig. 2.24).

Graph G_2 contains classes (c), properties (p), associations (as), instance specifications (is) (i.e., links and objects), slots (s)[44], literal strings (ls), and instance values (iv)[45]. These elements are first-class elements in the UML approach, i.e., classifiers of model elements. This is a contrast to the DSL approach, where objects, links, slots, and values are instances of

[44] To better distinguish instances of class *Slot* the according nodes have a dark background color.
[45] *LiteralString* and *InstanceValue* are subtypes of *ValueSpecification*; cf. Sect. 2.3.3.

instances of metaclass, metaassociation, and metaproperty respectively. Therefore, objects and links of UML-based models are translated into instances of the classifiers *is*, *s*, and *iv*, whereas slots and values are translated into instances of *s* and *ls* (cf. left-hand side of Fig. 2.13 that depicts the abstract syntax of a quite similar model). Note, that much detail has been omitted on the right-hand side of Fig. 2.26, e.g., morphism arrows and elements in G_2 and G_3.

We will not go into detail discussing Fig. 2.26. The examples show that the same concepts—typed, attributed elements and relationships between elements—can be realized in different domain-specific languages—DSL approach, UML approach, category of graphs and total graph morphisms—and that concepts can be translated between these languages. Furthermore, we showed that models of both the DSL and UML approach can be mapped onto graphs and that mapping even small models to labeled, typed graphs results in graphs containing many elements (i.e., nodes and edges). Even though these graphs are more fine-grained they can be used as formal system providing formally founded approaches with precisely defined semantics to realize formally founded models.

2.8.3. Realizing Model Transformation with Graph Transformation

Model transformations can be realized by graph transformations [TEG+05]. There are different approaches of graph-based transformation systems [FMRS07, CHM+02, dLV02]. One of these approaches is called *Story Driven Modeling (SDM)* [Zün01] which is implemented in the CASE tool Fujaba [NNZ00, FNTZ00]. SDMs are applicable in the domain of MDE to realize model transformation [GGZ+05].

Story driven modeling uses the concrete syntax of UML activity diagrams, object diagrams, and interaction diagrams. The "backend" of SDMs is formally defined relying on a 1st order logic-based approach to graph grammars defined in the programmed graph replacement system PROGRES [Sch91]. A unidirectional model transformation rule is defined in a *story diagram*—similar to UML activity diagrams—which defines a control flow between graph transformation rules. This allows to react on the result of a match, i.e., whether a match was found or not. A story diagram typically consists of *activities* and *transitions* that represent a certain control flow. An activity either contains a story pattern or a piece of Java code which is executed during runtime whenever the control flow enters the activity. Transitions are used to connect certain activities. They may have guards which constrains the control flow between a set of activities. In addition, a story diagram control structure called *for-each activity* allows to handle multiple matches of a pattern. Thus, a small graph transformation system is composed of basic graph transformation rules inside a story diagram. Story diagrams are used as implementation of methods. So, if a story diagram is executed it always runs in the context of a class (static method) or an object (non-static method). Additional context is passed to a story diagram as method parameters. Story diagrams may be used to define some of the dynamic semantics of language models by giving precisely defined meaning to a name of a method which transforms instances of the language model into another state that conforms to the syntax of the language model.

We decided to base our approach of bidirectional model transformation on SDMs. Therefore, we use the meta-CASE tool MOFLON (cf. Sect. 7.1) which is based on the CASE tool Fujaba and features SDM diagrams. From our bidirectional model transformation specification (cf.

Sect. 4.3) we derive unidirectional operational transformation rules and use these operational rules in our language translation framework (cf. Sect. 4.6). Finally, executable Java code is generated from SDM diagrams which implements the specified model transformation. In the following we will give a short overview of the elements of SDM diagrams which are used in later chapters in derived operational rules.

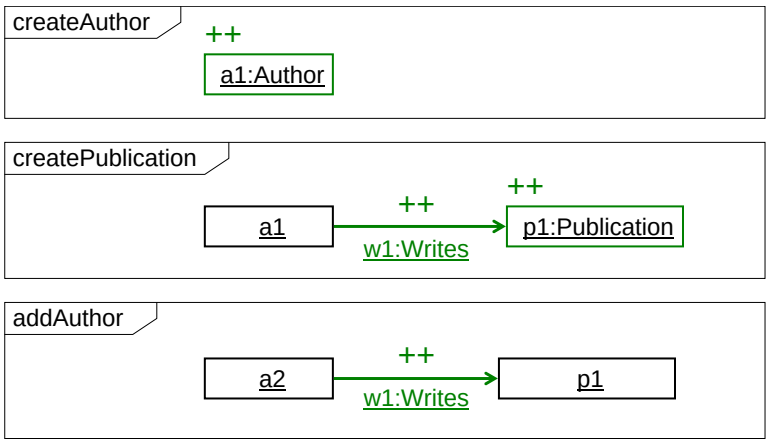


Figure 2.27.: Schematic view of story patterns.

The graph transformation rules are called *story patterns* and are visualized as a combination of object- and interaction diagrams. Figure 2.27 depicts three story patterns based on the library language model (cf. Fig. 2.25) that are similar to the productions depicted in Fig. 2.19[46]. Story patterns look up a model for a certain pattern in a given context. If the pattern is matched the model is finally transformed. This is analogous to the modification of a graph by a graph grammar production.

Story patterns use a shorthand notation for graph productions which summarizes the left- and the right-hand side of the underlying production. Instead of depicting both LHS and RHS as two separate parts of a story pattern, both sides are merged. The elements contained in the LHS and the RHS of the production, i.e., the context elements $L \cap R$, are denoted as black elements without any additional markup. The elements contained in the RHS only, i.e., elements $R \backslash L$ that are created, are denoted as green elements with an additional "++" or «create» markup. The story pattern "createAuthor" depicted in Fig. 2.27 creates a new author. The author is *bound* to a variable $a1$ of type *Author*. The variable is visible to all other story patterns in the scope of the execution of the story diagram. Pattern "createPublication" uses an already bound author $a1$ and creates a new publication $p1$. A "bound object" is denoted by only depicting the identifier $a1$ of the object, i.e., without showing a colon followed by the name of its classifier. In addition, "createPublication" links the new publication to the author indicating that the author has written the publication. Finally, the third pattern "addAuthor" links a formerly bound author $a2$ with a formerly bound publication $p1$.

[46] The notation of models uses arrow heads at the ends of a link to depict navigable ends of the classifying association (e.g., association *Writes*' navigable member end *publication*). This is contrary to our notation of graphs that represent models as introduced in Sect. 2.8.1 (cf. Fig. 2.24) where arrow heads are only used to denote edges.

Story patterns support additional graph grammar features that are used throughout this thesis and are not depicted in Fig. 2.27. They allow to specify deletion of elements, i.e., elements $L \backslash R$ that are contained in the LHS only. Elements to be deleted are denoted as red elements with an additional "−−" or «delete» markup. In addition, elements may be marked as optional which is indicated by a dashed border. Therefore, an element can be denoted to be optionally created, deleted, or matched. Negative application conditions (NACs) are also supported. An element marked negative (the element is crossed out (cf. Figs. 3.4 and 3.5 in the next chapter) must not be present during a match of the pattern. It can be thought of as a precondition of a pattern that must be met in order to apply the pattern.

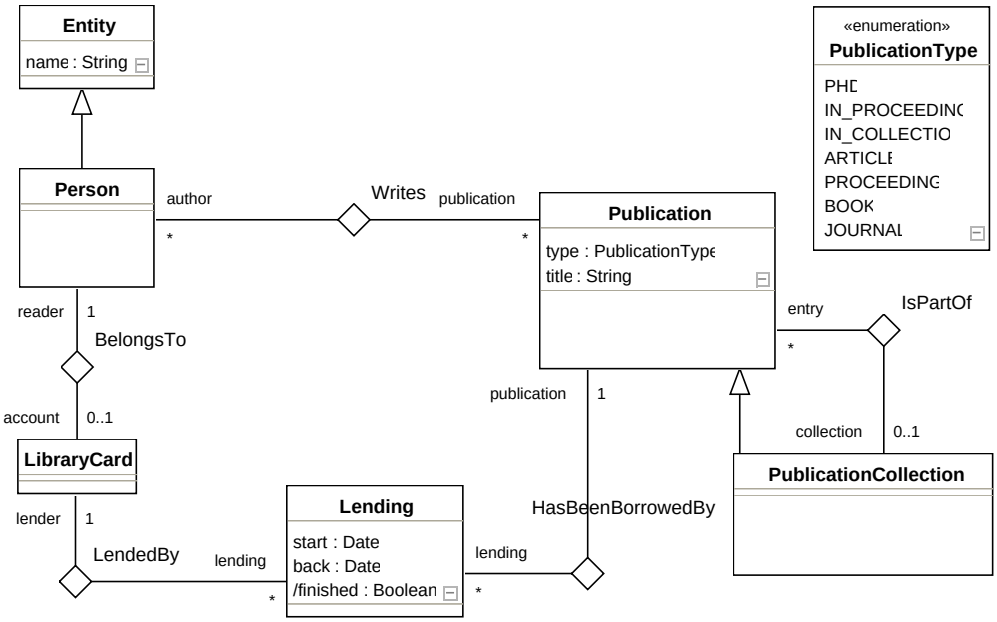


Figure 2.28.: Advanced library language model.

Story patterns require a language model, e.g., a class diagram, that defines the elements of the language. These elements can then be used in story patterns to define model transformations for the elements of the language. The transformations are then applied to models of the language. Figure 2.28 depicts an advanced version of the library language model introduced in Fig. 2.4. Note that we slightly changed the design of the library language model. The class *Author* has been renamed to *Person* and *author* is now used as role name (i.e., member end of an association) in conjunction with the association *Writes*. This allows to reuse *Person* in other situations, e.g. when it plays the role of a reader in conjunction with lending books from the library. Therefore, the name of a person being a very basic property of entities has been moved to its superclass *Entity*. As "publication" is a very general term, instances of publication are now categorized by their type (i.e., a literal of enumeration *PublicationType*). A publication is either a PHD thesis, a paper that appeared in the proceedings of a conference, a book, a part of a book (enumeration literal *IN_COLLECTION*), or an article that appeared in a journal. In addition, a publication may also be an instance of *PublicationCollection*—a

collection of smaller publications—like the proceedings of a conference that contains some papers, a book that is composed of some *IN_COLLECTION* publications, or a journal which consists of articles. Furthermore, a customer of the library, a reader, may now get an account if he likes to borrow a publication from the library. Then a library card associated with the reader is prepared where every publication that is lended by a reader is logged, i.e., a *Lending* instance is added to the library card and linked to the lended publication.

Fig. 2.29 depicts a screenshot of a story diagram which contains some more of the elements and features of the SDM language. It depicts a model transformation rule that defines the semantics of lending one book to a registered reader in the context of the library system.

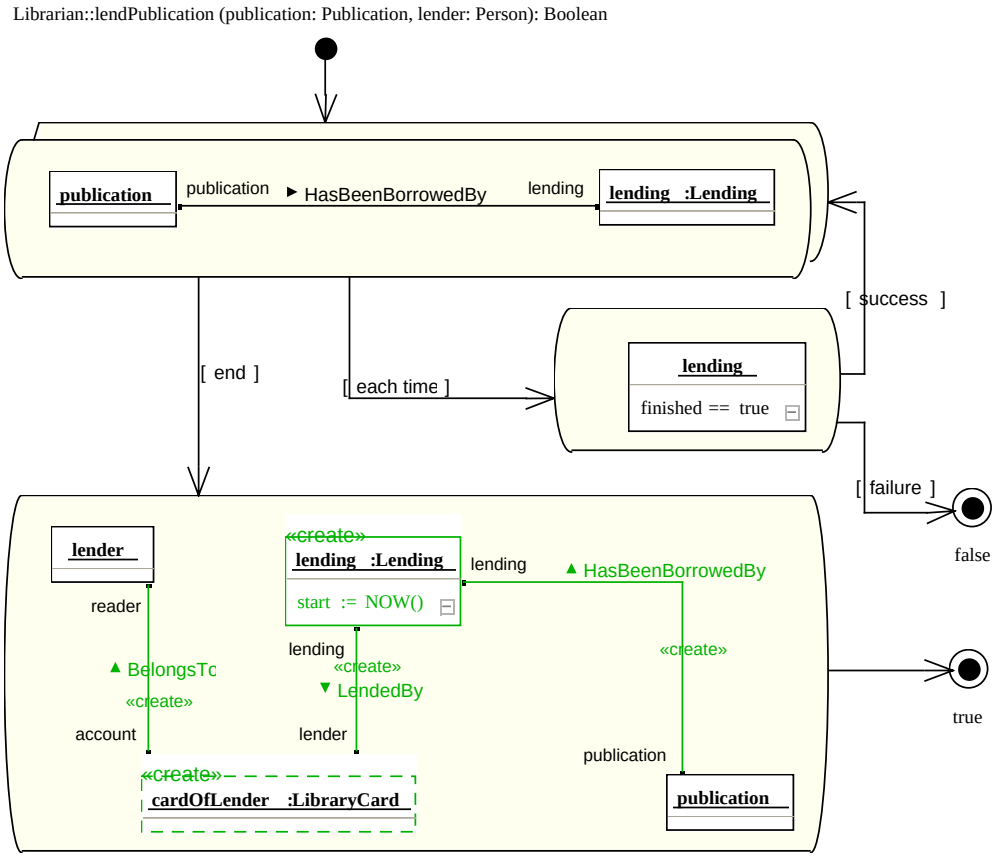


Figure 2.29.: Story diagram that models “lending a book to a reader”.

The entry point and the start of the control flow of a story diagram is its start node which is connected to a method declaration. The declaration of the method appears at the top of the start node. Story diagram 2.29 is attached to the method “lendPublication” contained in the class *Librarian*. The method has two parameters which add additional context to the story diagram. Therefore, the objects *publication* of type *Publication* and *lender* of type *Publication* are bound to the story diagram’s context. In addition, the declaration states that the story diagram must return a boolean value. Story diagram 2.29 consists of the start node, three activities, and two stop nodes. These elements are connected via transitions. Some of the

transitions have guards that constrain the control flow, i.e., a control token only follows a constrained transition if the guard is fulfilled.

Each activity contains a story pattern. The story pattern of the *for each* activity directly connected to the start node determines every lending occurrence in the context of the given publication object. Therefore, every match of the story pattern in the given context (i.e., *publication*) is computed, i.e., the object *lending* is (re)bound every time a new match is found. Then, the pattern contained in the activity connected via the [each time] transition is invoked. This pattern then in turn checks whether the matched lending is already finished. If this is not the case the story diagram is aborted via the [failure] transition. This way a precondition of the main activity of this story diagram is realized which ensures that a publication can be lended only if it is present in the library, i.e., not currently lended to another reader. After each match has been successfully checked, the main activity is now entered via the transition with the [end] guard. The story pattern of the main activity checks whether the lending person already has an account in form of a library card. If this is not the case, a new library card is created and associated with the lending person. This behavior is realized by the *optional create* construct depicted as object with a dashed border marked as create. Furthermore, a new lending entry for the given publication is created and associated with the library card. Finally, the start date of the lending is set to the current date. The lending entry indicates that the publication is currently—or has once been—lended. If the reader returns the publication, another operation then sets the return date so the lending will be marked as finished. Another person is then able to lend afresh the publication. The main activity hands control to the stop node that returns the value "true". A stop node leaves its story diagram and returns control to the caller which is, e.g., a transformation tool that has been triggered by a real librarian.

3. Integration of Formal Languages

Let's consider now the problem of translating between two computer languages.

[Hofstadter [Hof99, p. 380]]

In the preceding chapter we presented the fundamental concepts required for realizing model-based integration of formal languages. Now, we will discuss issues of language integration in order to finally realize bidirectional translators. The term "*integration*" is used throughout this thesis to indicate that two models, consisting of a number of elements, are somehow mapped in order to bidirectionally translate between them. Both models may still further exist on their own. The approach presented in this thesis uses model transformation and translation concepts to integrate languages and to realize bidirectional formal language translators. We start with an analogy of a natural language translation scenario in Sect. 3.1. Section 3.2 then discusses the process of language translation based on this analogy. Subsequently, a popular example of formal language integration—the integration of class diagrams and database schemata—is introduced in Sect. 3.3. Thereafter, the similarities between natural language translation and formal language translation are sketched in Sect. 3.4. Then, we discuss challenges that arise when realizing bidirectional translators in Sect. 3.5. We conclude this chapter with Sect. 3.6 by discussing the main ideas of model-driven integration and our model-based bidirectional formal language translation approach in particular.

3.1. CAB: A Natural Language Translation Analogy

The following scenario introduces common situations that occur during natural language translation. In the next sections we will see that they also arise during formal language translation.

Alex, an author, writes a publication about "How to Play Guitars" in a certain language. As a human being, Alex is capable of "speaking" at least one language—his native natural language: english. Let's imagine Alex produces his publication based on—or expressed in—the english language. Then the publication can be regarded as a model (model of thoughts of the author on how to play guitars) which is a "sentence" of the english language (more precisely the publication consists of many sentences). In general everything can be regarded as model in the sense of the definition of *model* in Sect. 2.2.1. "Reality" is perceived by humans as token model because humans filter information from reality (by their organs of perception) until the "perceived reality" finally arrives "in the brain". Even thoughts of a human being that are put down on paper are only models of the thoughts of their author. Typically an author writes his publication in his native language because the quality of the publication, e.g., syntactical correctness and style (i.e., written in "perfect" english) depend on the linguistic capabilities of

the author. In addition, the quality depends on the knowledge which the author has gained in the domain he is writing about. It is questionable whether an author writing about "How to Play Guitars" never having played a guitar may produce a good contribution.

The publication has an intention, i.e., transports a message from the author to the reader. Here, the message is how to make music with a guitar. The publication is expressed in a domain-specific language $DSL_{english,guitar}$ which is a combination of the english language and the terms used to describe everything related to playing guitars (music theory, parts of guitars, playing techniques like *bending* and *sliding*, etc.). Alex has developed an own internal specific domain language model $DLM_{english,guitar,Alex}$ of $DSL_{english,guitar}$ (cf. Figs. 2.14 and 2.8) while studying the subject "guitars and how to play them". To ensure that the message of the publication is interpreted by the reader as intended by the author, Alex' *specific guitar DLM* is very close to the common domain language model $DLM_{english,guitar}$ of $DSL_{english,guitar}$. On the other hand, Alex might have chosen to use new terminology. For example the term "gliding" to refer to creating continuous transitions in pitch—instead of the more common term "sliding". Though, he decided not to confuse his readers.

An example of variation in two related languages is the usage of the letters "H" (german notation) and "B" (english notation) when referring to pitches in music. Both letters refer to multiples of the frequency 493,88 Hz, i.e., two tempered semitones above A[1]. Originally, "B" was used in the german notation as well. But somewhen it became common to also use "H" instead of "B" when referring to this special pitch. One explanation why "H" is used instead of "B" in the german notation is given by [RE67]. In the 12th century a new feature was added to music. "B" was split into a higher pitch two semitones above A (B *durum* denoted as $\natural$) and a lower pitch one semitone above A (B *molle* denoted as b). When sheets of music were mechanically reproduced with early printing presses in the 16th century, the character "H" was used in movable type systems due to the visual similarity of the (handwritten) symbol $\natural$ and the character "H". Hence, it became common to use the notation "H" (instead of "B (durum)" or $\natural$) to refer to the 7th pitch of the *major scale based on C*, which became the basic scale since Zarlino (1571), in Germany.

Under normal circumstances this renaming of "H" to "B" would not matter at all. But unfortunately "B" is used to refer to the frequency 466,16 Hz[2] in the german notation—whereas "B_b" (speak: *B flat*) is used for this frequency in the english notation. So, the meaning of "B" in the german notation collides with the meaning of "B" in the english notation. This might lead to confusion because two musicians first have to agree on whether they are talking in the german or english notation when saying "B" (either the frequency 493,88 Hz or 466,16 Hz). This ambiguity is either resolved by explicitly stating the context in which "B" is used (german or english notation) or even prevented by only using the non-ambiguous notations "H" and "B_b". Alex discusses the above mentioned issue in his publication and decides to use the english notations "B" and "B_b".

Let us now assume, Chris, a reader of "How to Play Guitars" enjoyed this publication and wants to translate it into another language. She plans to translate it into her natural language, german, in order to make it easier for germans to get access to the contents of the publica-

[1] $f(2) = 440Hz * 2^{2/12}$ in *twelve-equal temperament* tunings with $A = 440Hz$.
[2] $f(1) = 440Hz * 2^{1/12}$ in *twelve-equal temperament* tunings.

tion, i.e., she wants to translate the book written in $DSL_{english,guitar}$ into $DSL_{german,guitar}$. Hence, Chris plays the role of a translator and decides to produce another publication that should correspond to its original. Therefore, she asks Alex if he likes to be involved into the translation process because he is the one who knows how to interpret[3] the original intention of the publication best. In addition, she decides to use the computer-based translation system Binaltas[4] which acts as a bidirectional sentence translator. It is capable of "speaking" both german and english, i.e., has internal language models of german and english. Moreover, it is able to improve its knowledge in both languages (e.g., by learning new words) and even to learn new languages. But, Binaltas does neither know anything about how to play guitars nor is it able to automatically resolve ambiguities existing in sentences of the input language. However, Binaltas is capable of reporting ambiguities to Chris who then in turn will ask[5] Alex so he might tell her about the intention he had when writing the ambiguous sentence in the original publication. She will then choose from a list of alternatives given by Binaltas in order to produce a translation that corresponds to its original. Binaltas in turn keeps track of the relations between the words in the original publication and the translated version to be able to trace the counterparts of sentences in the related publication.

Ambiguities arise in natural language sentences very frequently. Moreover, natural languages are informal and it is difficult to express things precisely, allowing to interpret different meaning than originally intended. Here the title "How to Play Guitars" may be translated in different ways. It might be interpreted as (a) "Wie man Gitarre spielt" or (b) "Wie man Gitarren spielt" both having different meaning and different degree of probability. The meaning of (a) refers to "How to make music using the instrument of type *guitar*" let's say with a chance of 5%, whereas (b) has the meaning of "How to play specific types of guitar" (e.g., 'Stratocaster' or 'Les Paul')—in order to get special sounds in combination with specific amplifiers and effects—with a chance of 95%. Binaltas detects these ambiguities and asks Chris whether to use (a) or (b). It suggests to use (b) because the probability of (b) is greater than (a). Instead, Chris comes to the conclusion that (a) is the right choice because she is aware of the book's content and the intention of both (a) and (b) in the german language. She shortly discusses this issue with Alex who internally marks the title to be reviewed in the next version of his publication. Binaltas continues translating the remaining part of the publication in conjunction with Chris and Alex which of course have to resolve further ambiguities. Moreover, Chris decided to use the german notations "H" and "B" and, therefore, has to teach Binaltas to automatically detect such notations and translate them appropriate. Finally, after completing the translation process, Chris publishes the translated version of "How to Play Guitars".

Later, Alex—who was inspired by the discussions with Chris—decides to review his initial version of "How to Play Guitars". He applies some changes, e.g., he changes the title to "How to Play Guitar". Binaltas—who still monitors activities in both publications—notices this change and informs Chris so she might decide whether this change affects the translated version. In this case, Chris does not need to react on the change as Alex did not change the intention of the title. But, Alex continues his review and corrects typos, adds some new

[3] In general the author of a book becomes a reader of his own book somewhen, having to reinterpret the original intentions he had when writing the book.

[4] **bi**directional **na**tural **l**anguage **t**ranslation **as**sistant (Binaltas, IPA:|bamæltəs|)

[5] Binaltas also assists in communication processes between Chris and Alex.

chapters, rephrases some sentences, deletes outdated material, and resolves ambiguities he was not aware of earlier. Happily, Binaltas assists Chris in managing to update the translated version incrementally so she might produce a new version that is again consistent with the new version in the original language.

3.2. Relationships in Translation Processes

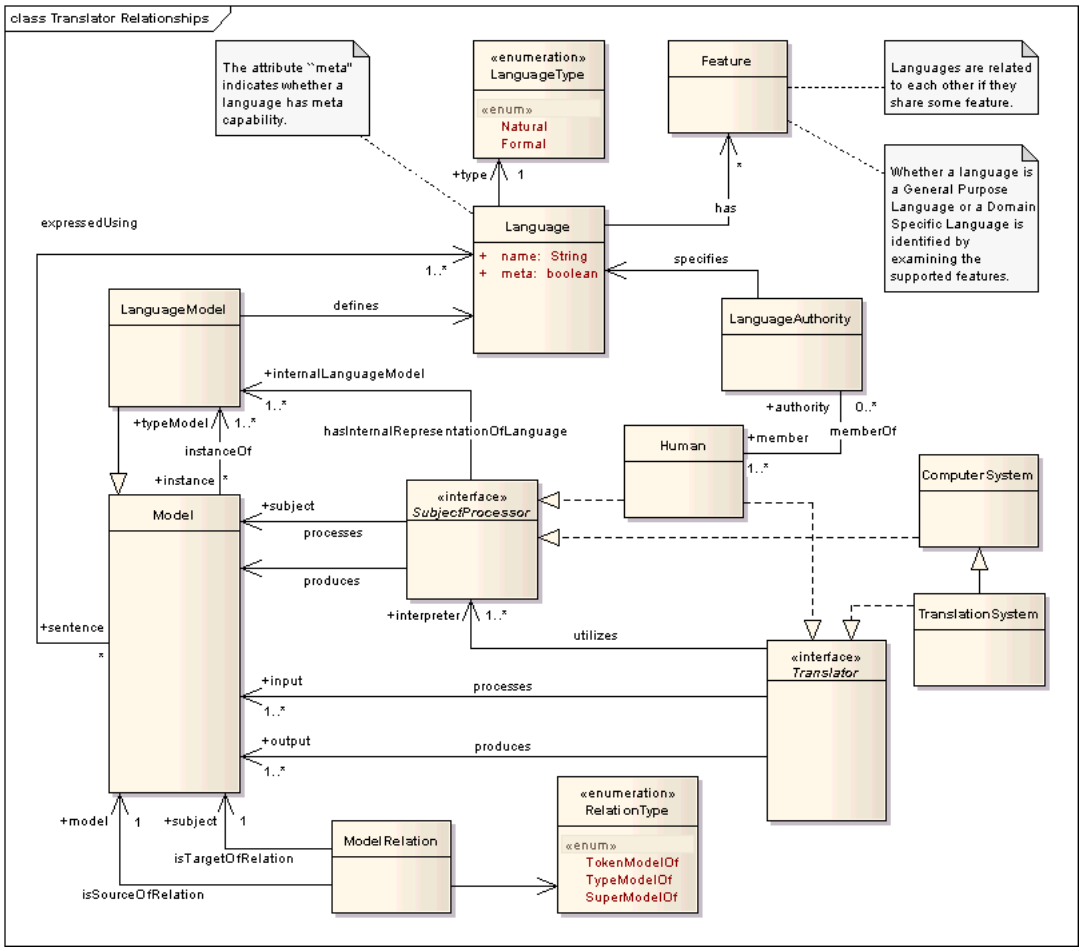


Figure 3.1.: Relations between components involved in a translation process.

Figure 3.1 depicts a domain model of the structural dependencies of the situation sketched in the Chris-Alex-Binaltas (CAB) example. It shows that humans as well as computer systems are able to process subjects (cf. interface *SubjectProcessor*) and to produce models of these subjects (cf. Sect. 2.2.1). Such processors will produce models if they have an intention for doing so. Examples of models that are produced in the CAB example are the publication and its translation, but also communication models such as emails exchanged between Chris and Alex—this was not explicitly mentioned in Sect. 3.1. Everytime Chris gets an email from

Alex (e.g., an answer to a question that arises during translation) she processes this email and produces another email containing her thoughts on Alex' answer.

Translators—humans or computer-based translation systems like Binaltas—produce an output model given at least one input model. Therefore, a translator utilizes a number of interpreters—either human beings or computer based systems—that "reason" about the intention of the (part of the) input model given to them by the translator and produce a translated version of this model in the desired language—if the interpreter does not face any problems during translation, like Binaltas who is not able to translate ambiguous models. To produce the output, the translator somehow coordinates the interpretation results (this is done by Chris in the CAB example). A translator does not need to interpret the given input model(s) himself, e.g., if it concentrates itself on coordinating the results produced by the utilized interpreters. However, if a human takes the role of a translator he might utilize himself as interpreter and no additional interpreters. This is only suitable if he is capable of speaking all languages of all input models he needs to process throughout the translation task in order to produce a corresponding version of the to-be-translated model. Typically, the "reasoning process" of a computer system is coded by a human software system engineer.

In general, an output is producible if the languages of the input model and the output model have something in common. A language supports a number of features and two languages are said to be related to each other if the languages share some features. General purpose languages typically provide features such as boolean logic, arithmetics, generics, ability to make statements, modularization concepts, and refinement. Features of domain-specific languages—like the library language discussed in Sect. 2.2—are, e.g., the ability to express facts (e.g., a book with a specific title which is written by an author) and activities (e.g., a book contained in a library is lended by a reader) occurring in the domain. In particular, a translator will only produce an output if the utilized interpreters "speak" the input and output languages, i.e., they have internal representations of these languages. Such an internal representation is a language model (e.g., Alex' internal domain language model $DLM_{english,guitar,Alex}$) which is typically a variation of a precise and complete language model that defines the language (e.g., $DSL_{english,guitar}$).

For a description of the classes *Language*, *Model*, and *LanguageModel* and their relationships, which are depicted in Fig. 3.1, we refer to the discussion in Sect. 2.2.4 (cf. Fig. 2.8). The class *ModelRelation* is a directed relationship which allows to relate two models, e.g., to state that one model is the token model of another model—its subject—(cf. Fig. 2.5 in Sect. 2.2.3). The *instanceOf* relationship between *Model* and *LanguageModel* has been explicitly modeled as association. It might be realized by an instance of *ModelRelation* of type *TypeModelOf* as well. But association *instanceOf* might be regarded as a *derived association* which abbreviates navigation from a model to its type model.

A language authority creates and specifies new languages. Examples of such authorities are the OMG[6] which has specified modeling standards such as UML, MOF, QVT, and OCL. The World Wide Web Consortium (W3C)[7] is another language authority which develops standards (e.g., HTML, XML, and XSLT—a language for transforming XML documents) to ensure the

[6] `http://www.omg.org`
[7] `http://www.w3.org`

long-term growth of the Web. Authorities for the written german language are, e.g., the Duden[8] or the Wahrig[9], whereas [Per08] and [SHE06] are authorities for the written english language. But even more unofficial language authorities are imaginable, like companies or specific groups of people that produce their own proprietary domain-specific languages.

3.3. Integrating Class Diagrams and Database Schemata

Based on the natural language translation analogy (the CAB example) we will now address the question of how to build Bifaltas[10]—one of the siblings of Binaltas—throughout the rest of this thesis. That is, building (semi-)automated bidirectional formal language translators.

Throughout this contribution we will discuss integration of formal languages referring to mappings of class diagrams (*object domain*) to relational database schemata (*data domain*) and vice versa. This integration scenario is abbreviated as *CDDS*. Both languages are well-known to software engineers and share some features and concepts because they both belong to the domain of *relational data models*. This example is quite popular and discussed in numerous publications of the model and graph transformation community [Bru06, Obj08] as well as in the database community [Amb03]. It can also be regarded as an official benchmark for QVT related approaches [Kön09] that are used to realize unidirectional or bidirectional mappings between two languages. We adopt this example and slightly modify it such that it (a) covers typical situations when integrating formal languages and (b) we are able to discuss arising problems and some solutions which are presented in this contribution.

The main idea behind integrating the class diagrams language L_{CD} and the database schemata language L_{DS} is to map persistent classes and database tables onto each other. For example, if the library language model is specified as class diagram expressed using L_{CD} this would lead to the situation depicted on the left-hand side in Fig. 2.26, where L_{CD} plays the role of the UML 2 Superstructure. Then, the class *Author* and its attribute *name* are related with[11] a table named *Author* and a column named *name* of an SQL based relational database management system (RDBMS) which is utilized by the library. Note that instances of a class (e.g., objects representing authors) and entries in a database table (i.e., rows of a table) are not regarded in CDDS. That is, models containing the data of a library (i.e., object diagrams) and database entries are not mapped onto each other. This would require an extension of the language models and additional mappings between instance specifications (i.e., objects and links) and database entries. But this is not discussed in this thesis.

It is common practice to call the domains of the languages involved in a translation process *source domain* and *target domain*. When talking about bidirectional translators one fixes these roles and then distinguishes between *forward translation* and *backward translation*—instead of

[8] `http://www.duden.de`

[9] `http://www.wahrig.de`

[10] **bi**directional **f**orm**a**l **l**anguage **t**ranslation **as**sistant (Bifaltas, IPA:|baɪfæltəs|)

[11] Note that we use the phrases "to relate sth. with sth." and "to map sth. and sth. onto each other" to refer to a bidirectional mapping. The phrases "to relate sth. to sth." and "to map sth. to sth." are used to refer to a unidirectional mapping.

reassigning the source and target roles. We assign the source role to class diagrams and the target role to database schemata.

3.3.1. Syntax of CD and DS Language Models

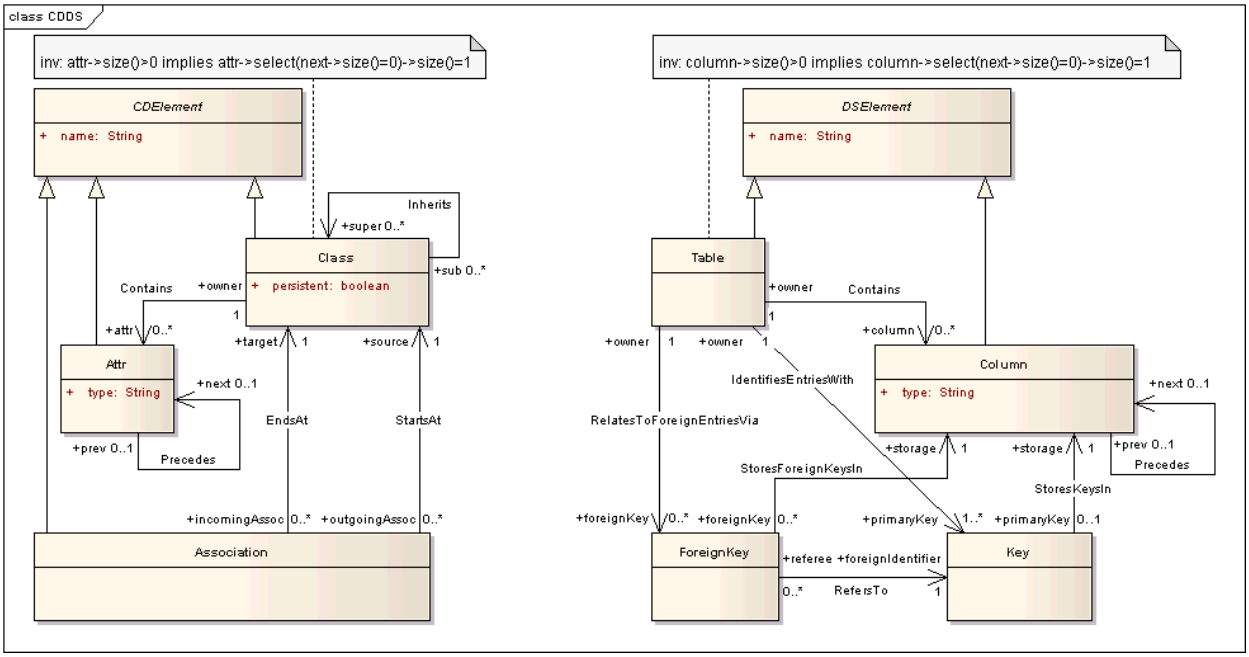


Figure 3.2.: Language models of simple class diagrams and database schemata.

The language models of class diagrams LM_{CD} and database schemata LM_{DS} used in the integration example are depicted in Fig. 3.2. It is important to notice that LM_{CD} is a simplified language model of class diagrams and neither the class diagrams part of the UML nor the MOF. Consequently, LM_{CD} varies in some points compared with UML and MOF class diagrams. If we are talking about integration of class diagrams and database schemata in the following we always refer to LM_{CD} and LM_{DS}.

The syntax and static semantics of both language models LM_{CD} and LM_{DS} are expressed using the MOF and OCL. We will first examine the structural features provided by the languages. Both languages feature the concepts “types”, “properties”, and “relationships”.

The left-hand side of Fig. 3.2 depicts a language model of class diagrams. It defines classes, primitive attributes, and directed associations. Classes may inherit features of other classes which is indicated by the subclass to superclass relationship *Inherits*. As a special feature, multiple inheritance is supported so each subclass may have zero to many superclasses. Classes may be marked as persistent with the boolean flag *persistent*. Furthermore, classes may contain zero to many attributes which are ordered by the successor/predecessor relationship *Precedes*. A successive attribute is identified by the role *next* and a preceding attribute by the role *prev*. Attributes are always primitive and its primitive datatype is encoded in the value of property *type*. Relations between classes are realized by directed associations.

The language model of relational database schemata is depicted on the right-hand side of Fig. 3.2. It defines tables, columns, (primary) keys, and foreign keys. Each table has at least one primary key which is used to identify entries in this table (i.e., rows of a table—which are not part of this example). A table may be related to another table via a foreign key which connects to the primary key of a foreign table. This way, entries of a table can be related to entries in another table which is quite similar to the association concept in the class diagram domain that relates a class with another class. Furthermore, each table may contain a number of columns. Likewise to attributes, columns are ordered by the successor/predecessor relationship *Precedes*.

Note that structuring and modularization of elements in both language models is not part of the example in order to keep the example, especially models depicted later on, more compact. However, all instances of *Class* and *Association* depicted in one diagram are assumed to be contained in the same package, i.e., the class *Package*. Similarly, all instances of *Table* depicted in one diagram are assumed to be contained in the same database schema, i.e., the class *Schema*. Therefore, we specify OCL constraints that, e.g., check all classes contained in one package in the context of *Package* and then navigate to its owned classes via the association end *class*.

3.3.2. Constraints in CD and DS Language Models

Let us have a closer look at the constraints of the language models. Models must fulfill these constraints in order to be valid. Successors of attributes and columns are realized via the association *Precedes*. The multiplicity "0..1" of the *next* endpoint of the association denotes that each attribute and column may have a successor, but need not. These are multiplicity constraints that might be expressed by OCL invariants $inv_{CD:P:n:mult}$[12] and $inv_{DS:P:n:mult}$. In addition, the OCL invariants depicted in Fig. 3.2 constrain the number of elements without successor. Due to these constraints, a class containing attributes must have exactly one attribute that has no successor. Similarly, tables and columns are constrained by an analogous OCL invariant.

There are more invariants that must be fulfilled by instances of both language models. These are not depicted in Fig. 3.2 but given here in textual notation. There are still more OCL invariants that could be added to the language models. For example $inv_{DS:F:inheritance}$ in Sect. 5.2.1 and constraints that forbid inheritance cycles in LM_{CD}. The latter constraints are not discussed in this thesis.

$inv_{CD:Pkg:assoc:unique}$ Each association contained in a package must have a unique name.
context Package inv: assoc->forAll(a1, a2 | a1 <> a2 implies a1.name <> a2.name)

$inv_{CD:C:attr:unique}$ Each attribute contained in a class must have a unique name.
context Class inv: attr->forAll(a1, a2 | a1 <> a2 implies a1.name <> a2.name)

$inv_{CD:C:attr:first}$ If a class contains attributes then exactly one "first" attribute must exist (i.e., an attribute that has no predecessor).
context Class inv: attr->size()>0 implies attr->select(prev->size()=0)->size()=1

[12] $inv_{CD:P:n:mult}$ is a shortcut for $CD : Precedes : next : multiplicity$

$inv_{CD:C:attr:last}$ If a class contains attributes then exactly one "last" attribute must exist (i.e., an attribute that has no successor).
context Class inv: attr->size()>0 implies attr->select(next->size()=0)->size()=1

$inv_{DS:T:col:unique}$ Each column contained in a table must have a unique name.
context Table inv: column->forAll(o1, o2 | o1 <> o2 implies o1.name <> o2.name)

$inv_{DS:T:col:first}$ If a table contains columns then exactly one "first" column must exist (i.e., a column that has no predecessor).
context Table inv: column->size()>0 implies column->select(prev->size()=0)->size()=1

$inv_{DS:T:col:last}$ If a table contains columns then exactly one "last" column must exist (i.e., a column that has no successor).
context Table inv: column->size()>0 column->select(next->size()=0)->size()=1

In conjunction, the multiplicities of both ends of the association *Precedes* that relates attributes (i.e., $inv_{CD:P:p:mult}$ and $inv_{CD:P:n:mult}$) and the two OCL constraints $inv_{CD:C:attr:first}$ and $inv_{CD:C:attr:last}$ are able to detect violations of a strict total ordering of attributes. The same applies to the strict total ordering of columns in LM_{DS}. Violations are detected by constraints $inv_{DS:P:p:mult}$, $inv_{DS:P:n:mult}$, $inv_{DS:T:col:first}$, and $inv_{DS:T:col:last}$.

3.3.3. Producing CD and DS Models

In order to produce instances of both language models, production rules that create well-formed models are specified as story patterns in the SDM language. The patterns that build types, properties, relations, and inheritance structures in both languages are depicted in Figs. 3.3, 3.4, 3.5, and 3.6. Note that most of these patterns will be used later on Chap. 4 when creating a mapping specification that integrates class diagrams and database schemata according to the mapping requirements that will be stated in Sect. 3.3.5. Though, the depicted patterns are not yet a mapping specification!

Figure 3.3 (a) depicts the patterns "createPersistentClass", "createPersistentSuperclass", and "createPersistentSubclass" which are part of LM_{CD} and produce elements related to the type-concept of LM_{CD} that features multiple inheritance. Pattern "createPersistentClass" creates a persistent class $c1$ and sets the name of the class to a free variable n. This variable has to be specified by the caller of the pattern. A persistent superclass $c2$ is created by pattern "createPersistentSuperclass". The pattern requires another class $c1$ as context. It links the newly created superclass with $c1$ identifying the latter as the subclass of $c2$. Likewise, pattern "createPersistentSubclass" creates a new subclass $c1$ and links it to its superclass $c2$.

Pattern "createTable", depicted in Fig. 3.3 (b) creates a new table $t1$ and sets the table's name to the free variable n. In addition, it creates a primary key $pk1$ and a new column $o1$. The name of column $o1$ is set to the reserved column name *"_entryID"* which must not be used for other columns. The type of the column is set to *"NUMBER"* which indicates that numerical values will be stored in this column. Column $o1$ will contain the primary key values of table entries and allows to uniquely identify entries of a table. Finally, the three objects are linked with each other.

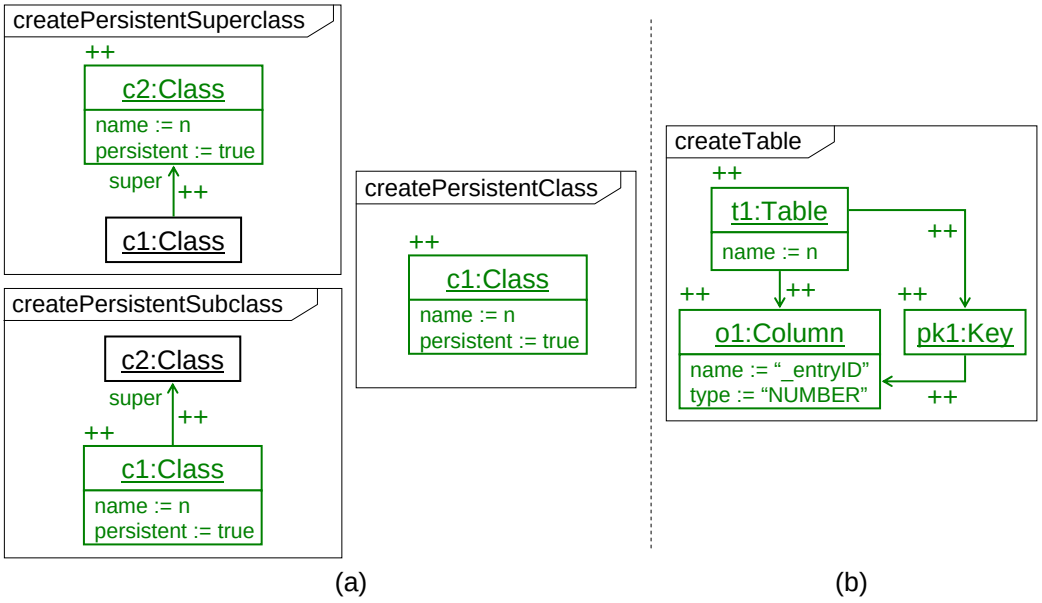


Figure 3.3.: Story patterns that produce types in (a) LM_{CD} and (b) LM_{DS}.

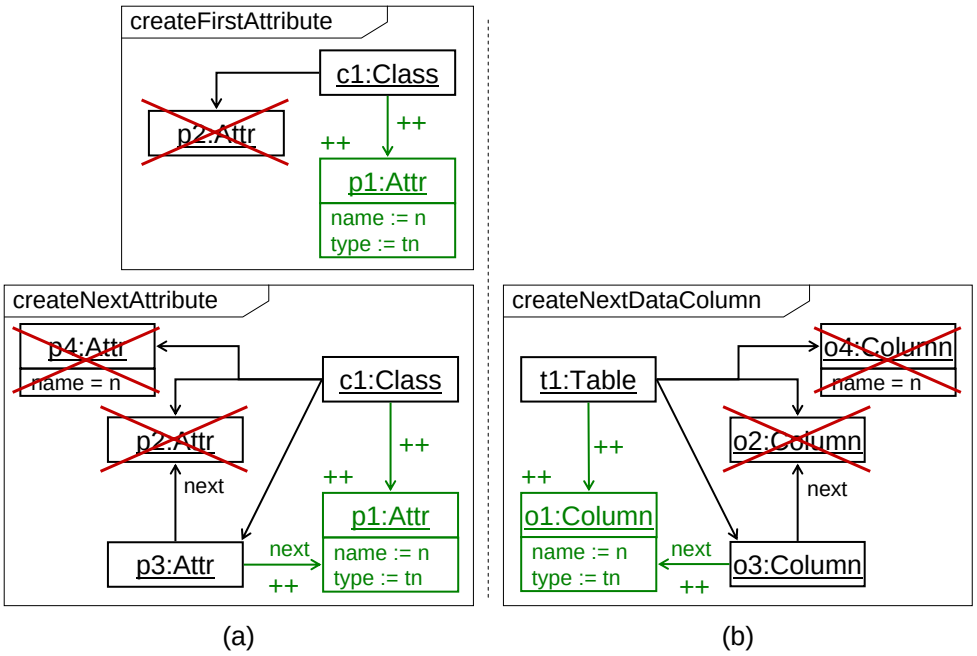


Figure 3.4.: Story patterns that produce properties in (a) LM_{CD} and (b) LM_{DS}.

Figure 3.4 (a) depicts two patterns that feature adding an ordered number of attributes to a class. Pattern "createFirstAttribute" is used to create the first attribute contained in a class. This attribute has the feature that right after creation time it has no predecessor and no successor. Due to the NAC this pattern is applicable only once—if the class does not already contain an attribute. So, this NAC ensures that at most one attribute is created per class that has no successor and, therefore, prohibit a violation of $inv_{CD:C:attr:last}$. Pattern "createNextAttribute" is applicable in situations where a class has at least one attribute, i.e., pattern "createFirstAttribute" has been applied earlier. The first NAC ensures OCL constraint $inv_{CD:C:attr:unique}$ holds after rule application, i.e., no attribute $p4$ with the given name n must be already contained in the class $c1$. Otherwise the pattern won't be executed. The second NAC ensures that the created attribute $p1$ is the successor of an attribute $p3$ that does not have a successor $p2$ yet. The effect of this NAC is that the pattern searches for the "last" attribute of the class and then makes the newly created attribute $p1$ the new "last" attribute. So, it ensures that the multiplicity constraint $inv_{CD:P:n:mult}$ of the endpoint "next" holds after application of the pattern. Moreover, it ensures that OCL constraints $inv_{CD:C:attr:first}$ and $inv_{CD:C:attr:last}$ are satisfied after rule application. That is, a valid ordering of attributes is achieved because—except of the "first" and the "last" attribute contained in a class—every other attribute of this class is linked with one predecessor and one successor.

Pattern "createNextDataColumn" creates ordered columns the same way as pattern "createNextAttribute" creates ordered attributes. No additional pattern is necessary that creates the first column in a table because this is already ensured by pattern "createTable". Consequently, the column which stores primary keys of table entries is always the first column.

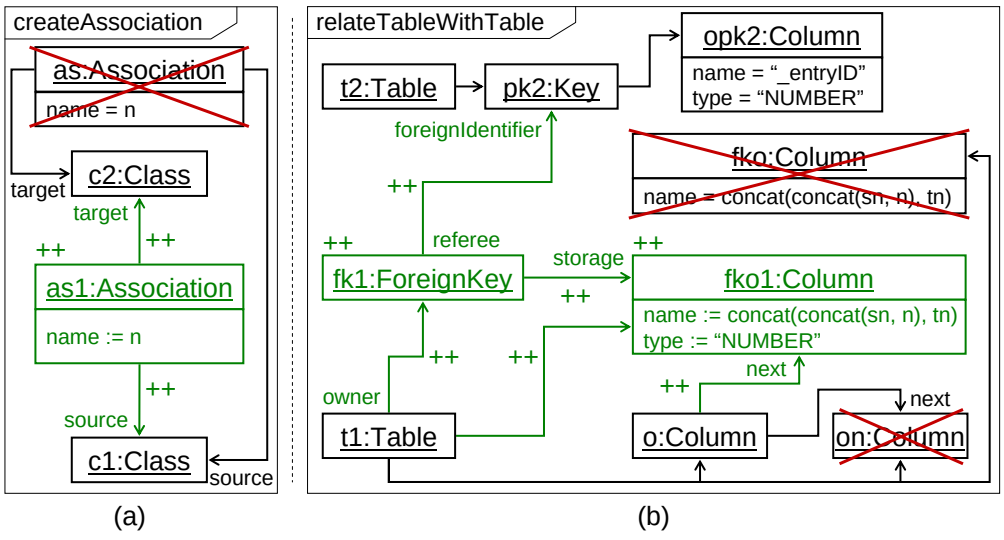


Figure 3.5.: Story patterns that produce relationships in (a) LM_{CD} and (b) LM_{DS}.

Figure 3.5 depicts two patterns that create relations in both domains. Pattern "createAssociation" creates a directed association which is represented by an instance of class *Association* using class $c1$ as its source and class $c2$ as its target. An association is only created if no other association with the same name already exists (cf. OCL invariant $inv_{CD:Pkg:assoc:unique}$).

Pattern "relateTableWithTable" defines a many-to-one relation between table $t1$ (the source type of the relation) and table $t2$ (its target type). This is due to the fact that entries of table $t1$ may reference up to one entry of table $t2$ with the newly created foreign key column $fko1$. Entries of table $t2$ may be referenced by zero to many entries of table $t1$ using the foreign key $fk1$ which refers to the primary key $pk2$ of table $t2$. The first NAC ensures invariant $inv_{DS:T:col:unique}$, i.e., that no other column exists in the table that has the name which is about to be given to the new column $fko1$. The result of this NAC is that the relation name n is not allowed to be used twice in the context of one specific table to refer to the same foreign table. The second NAC simply "searches" for the "last" column o of table $t1$. If the NACs do not block the application of the pattern, it creates a foreign key $fk1$ and links it to the primary key $pk2$ of the relation's target type. In addition, it creates a foreign key column $fko1$ which will contain for every entry in table $t1$ that is related to an entry in $t2$ a value in column $fko1$ which is equal to the value of the related entry in table $t2$ contained in the primary key column $opk2$. The relation's name n is encoded in the foreign key column $fko1$ such that the name of the relation's source sn (i.e, the name of table $t1$), the relation's name n, and the name of the relation's target tn (i.e, the name of table $t2$) are concatenated.

Note that the concatenation operation "concat" is invertible because it uses a unique reserved character as separator[13] when concatenating two strings. The operations "splitGetFirst" and "splitGetSecond" split a concatenated string and return the first and the second string respectively that has been passed to the operation "concat".

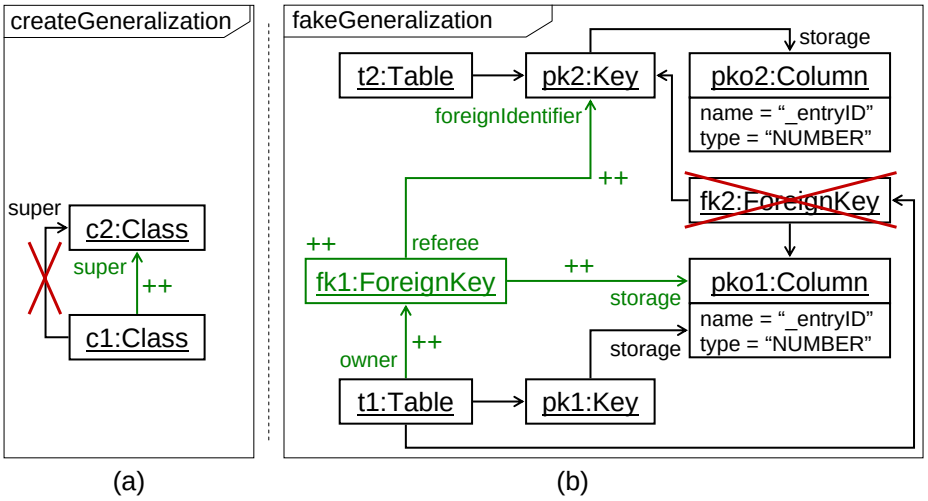


Figure 3.6.: Story patterns that produce generalizations in (a) LM_{CD} and (b) LM_{DS}.

Finally, Fig. 3.6 depicts patterns that create inheritance structures in both domains. Pattern "createGeneralization" establishes a new inheritance link between two already existing classes $c1$ and $c2$ if no inheritance relationship has been established already. The role of the subclass is applied to class $c1$, whereas the role of the superclass is applied to class $c2$. Pattern "fakeGeneralization" creates a generalization in LM_{DS} which is simulated by a foreign key. This approach to realize inheritance in databases is discussed in [Amb03]. The pattern creates

[13] Throughout this thesis we will use the character '@' as separator.

a new foreign key $fk1$ owned by the subtable $t1$ and connects this foreign key with the primary key column $pko1$ of subtable $t1$. In addition, the foreign key is linked to the primary key $pk2$ of supertable $t2$ that has been created by pattern "createTable".

Accordingly, the primary key column of a table is used both as primary key and foreign key if the table is a subtable contained in an inheritance structure. Moreover, the primary key of the *root table* of an inheritance structure is used by all its subtables. Therefore, in the case of a subtable, its primary/foreign key is used to maintain the relationship to its supertable up the inheritance structure to the root table. The same applies in case of multiple inheritance: the primary key is then used by multiple foreign keys and the relationship to all supertables up the inheritance structure to the root tables has to be maintained. Consequently, the ID of every table entry in an inheritance structure must be unique in the inheritance structure.

3.3.4. Examples of CD and DS Models

In the following we discuss models of class diagrams and database schemata. First, we will examine some invalid models which are depicted in Fig. 3.7. These models either do not conform to the syntax definition of their language model or they violate an OCL constraint. That is, the models are not well-formed (cf. Sect. 2.2.4).

Schema 3.7(a) (depicted in Fig. 3.7 (a)) contains a single table object. This violates multiplicity constraint $inv_{DS:Id:p:mult}$ specified in LM_{DS}, which demands that exactly one primary key—an instance of Key—must be linked to every table. The foreign key $fk1$ in schema 3.7(b) violates three multiplicity constraints. Foreign key $fk1$ is neither related to an owning table nor to a primary key that acts as a foreign identifier. Moreover, it is not associated with a column where the foreign key values are stored. Class c in package 3.7(c) does not violate a multiplicity constraint but it does neither fulfill $inv_{CD:C:attr:first}$ nor $inv_{CD:C:attr:last}$, i.e., ordering of attributes is destroyed. This is due to the fact that c contains two "first" and two "last" attributes which is forbidden by these constraints. Attribute $a1$ in package 3.7(d) is contained in two classes which violates multiplicity constraint $inv_{CD:Cont:o:mult}$ in LM_{CD}. The same multiplicity constraint is violated by the attributes $a1$ and $a2$ contained in package 3.7(e) because none of these attributes is owned by any class. Package 3.7(f) contains an association $as1$ which is linked to its source but the link to its target is missing. Therefore, the association is broken because every association must be linked to exactly two classes—source and target. Attribute $a1$ in package 3.7(g) violates multiplicity constraint $inv_{CD:P:n:mult}$ because it is linked to two successors. In addition, invariant $inv_{CD:C:attr:last}$ is violated because class c contains two attributes $a2$ and $a3$ which are "last" attributes. Finally, class c in package 3.7(h) violates both invariants $inv_{CD:C:attr:first}$ and $inv_{CD:C:attr:last}$ because class c contains attributes but neither a "first" nor a "last" attribute. Though, all attributes fulfill their multiplicity constraints.

Now, we are going to discuss valid (i.e., well-formed) models of CDDS. Package 3.8(a) which is depicted in Fig. 3.8 is valid according to LM_{CD}. It contains a single class c which was produced by applying pattern "createPersistentClass". Contrary to table t depicted in schema 3.7(a), class c does not violate any constraints in LM_{CD}. Therefore, it is well-formed according to the syntax of LM_{CD} as well as to its static semantics. Package 3.8(b) is also valid according to LM_{CD}. It contains a class and three correctly ordered attributes which were produced by applying patterns "createPersistentClass", "createFirstAttribute", "createNextAt-

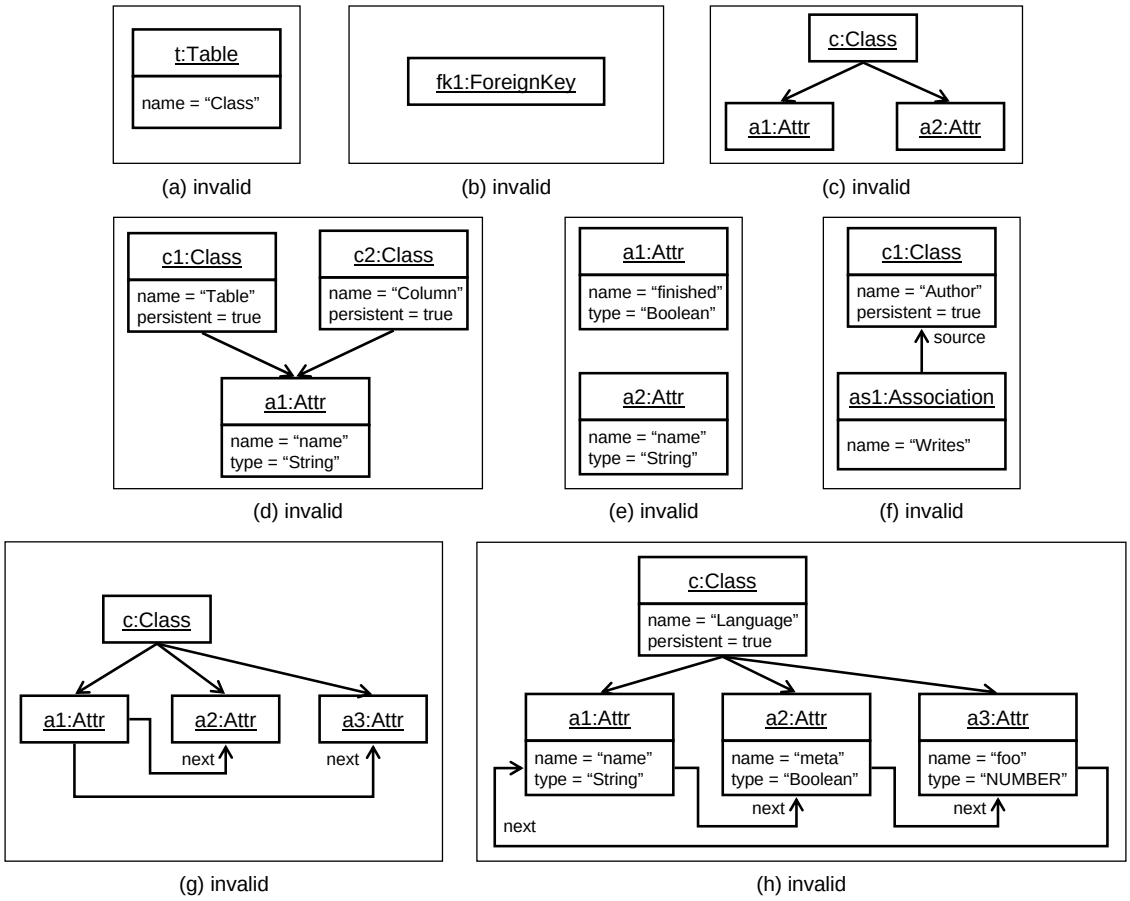


Figure 3.7.: Examples of invalid CD and DS models.

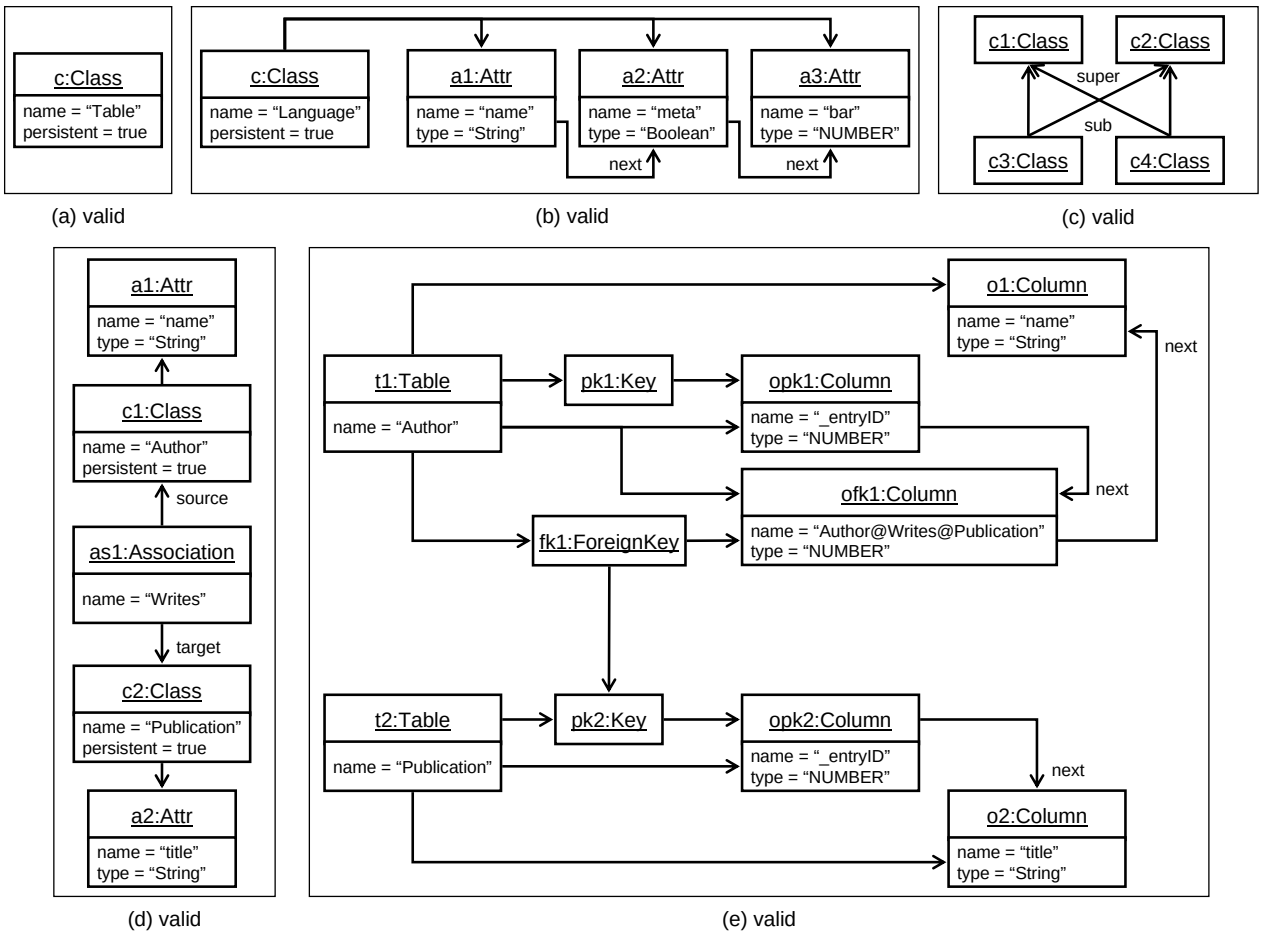


Figure 3.8.: Examples of valid CD and DS models.

tribute", and "createNextAttribute" in that order. A diagram which contains a valid example of multiple inheritance is depicted in package 3.8(c). It contains four classes. Class $c3$ and $c4$ both inherit from two superclasses $c1$ and $c2$. The elements of this package were produced by the patterns "createPersistentClass", "createPersistentClass" "createPersistentSubclass", "createPersistentSubclass", "createGeneralization", and "createGeneralization". But they may have been produced also by applying patterns "createPersistentClass", "createPersistentSubclass", "createPersistentSuperclass", "createPersistentSubclass", and "createGeneralization". Or even by applying "createPersistentClass", "createPersistentClass", "createPersistentClass", "createPersistentClass", "createGeneralization", "createGeneralization", "createGeneralization", and "createGeneralization". The order of pattern applications is not reproducible by examining the model. This example shows that there are rules which might be executed independently from another leading to the same result (in terms of the graph grammar world this is called "*confluence*"). Note that pattern "createGeneralization" is always required when producing this kind of inheritance relationship, i.e., patterns "createPersistentSubclass" and "createPersistentSuperclass" in conjunction with pattern "createPersistentClass" are not sufficient.

Package 3.8(d), which is valid according to LM_{CD}, contains two classes—each containing an attribute—which are related via an association. The diagram reflects the class diagram example of the library domain depicted in the upper left part of Fig. 2.4. The elements of package 3.8(d) were produced by the sequence of pattern applications $SEQ^{orig}_{CD:3.8(d)}$ = ("createPersistentClass", "createPersistentClass", "createFirstAttribute", "createFirstAttribute", "createAssociation"). The diagram may have been produced also by applying the patterns in the following order: "createPersistentClass", "createFirstAttribute", "createPersistentClass", "createAssociation", and "createFirstAttribute". Or even by the pattern sequence $SEQ^{alt}_{CD:3.8(d)}$ = ("createPersistentClass", "createPersistentClass", "createAssociation", "createFirstAttribute", "createFirstAttribute"). Finally, we discuss database schema 3.8(e) which is valid according to LM_{DS}. It contains a model that corresponds to the model depicted in package 3.8(d). Its original sequence of pattern applications is $SEQ^{orig}_{DS:3.8(e)}$ = ("createTable", "createTable", "relateTableWithTable", "createNextDataColumn", "createNextDataColumn"). This database schema realizes a many-to-one relationship from the author table to the publication table. Each author entry may reference one publication entry by using column $ofk1$. A publication entry in table $t2$ may be referenced by multiple author entries. Note the order $(opk1, ofk1, o1)$ of columns in table $t1$ and $(opk2, o2)$ of columns in table $t2$ which is defined by the links of type *Precedes*. The number of production sequence candidates is reduced dramatically for tables that have many columns due to the given order of *data columns* and *key columns*. This order informs about the moment of application of the patterns "createNextDataColumn" and "relateTableWithTable". Therefore, one of the impossible production sequences is $SEQ^{invalid}_{DS:3.8(e)}$ = ("createTable", "createTable", "createNextDataColumn", "createNextDataColumn", "relateTableWithTable") because then the order of columns in table $t1$ would be $(opk1, o1, ofk1)$.

3.3.5. Mapping CD and DS

Now we discuss which elements of LM_{CD} and LM_{DS} will be mapped onto each other in our example. In the general case there exists more than one possibility to define a mapping between two languages and each mapping solution has its advantages and disadvantages. We will apply some of the mapping proposals given in [Amb03] which discusses also different mapping approaches for class diagrams and database schemata. This section will serve as requirements document that is used in Chap. 4 when creating the CDDS mapping specification.

Sensibly, one package (i.e., class diagram) in the object domain and one schema in the data domain are mapped onto each other. Classes are related with tables, attributes with columns and associations with relations based on the foreign key concept. Ordering of attributes is related with ordering of columns. We decided to relate each class of a class hierarchy with a separate table. In our example the inheritance feature is supported directly only by LM_{CD} and not by LM_{DS}. That is LM_{CD} provides a language element—association *Inherits*—that is intended to realize inheritance. Therefore, we have to encode the additional structural information in the target domain (cf. pattern "fakeGeneralization" in Fig. 3.6 (b)) otherwise we would lose this information present in the source domain during mapping. Another mapping strategy would be to map the entire class hierarchy to a single table. We decided against the latter strategy because it would make the mapping specification much more complicated and does not bear any advantages except for faster access of data because the data is in one table (cf. discussion in [Amb03]).

The relationships (i.e., associations) *Contains* and *Precedes* are mapped *one to one*, i.e., each contains-link has one corresponding contains-link in the other domain. The relation of associations *StartsAt* and *EndsAt* in the context of the class *Association* with their counterparts in the target domain is more complex because it is *not a one to one mapping*. Their relation to the associations *RelatesToForeignEntriesVia*, *RefersTo*, *StoresForeignKeysIn*, *Contains*, and *Precedes* in the context of *ForeignKey* will be discussed in Sect. 4.4.

For reasons of simplicity, we assume that each instance of type *Association* in LM_{CD} is at most many-to-one, i.e., either one-to-one or many-to-one. That is, many-to-many associations are not directly supported in the class diagram system modeled by LM_{CD}. Consequently, if relationship *Writes* that relates authors with publications (cf. Fig. 3.8 (d)) is modeled as association based on LM_{CD}, *Writes* relates the source class *Author* with the target class *Publication*. Further, we assume that the source class of an instance of *Association* is always the "many class" and the target class is the "one class". Then *Author* is the many class and *Publication* the one class, i.e., association *Writes* is a many-to-one relation. Therefore, a publication may be related with many authors but an author contained in this system could be related with only one publication.

However, if a many-to-many relationships is required, it may be converted into two many-to-one relationships involving an *association class* [Obj09b] in the object domain or *associative table* in the data domain; [Amb03] states:

> *There are two ways to implement many-to-many associations in a relational database. The first one is to implement in each table the foreign key column(s) to the other table several times. [...] A better approach is to implement what is called an associative table, [...], which includes the combination of the primary keys of the tables*

> *that it associates. [...] The basic "trick" is that the many-to-many relationship is converted into two one-to-many relationships, both of which involve the associative table.*

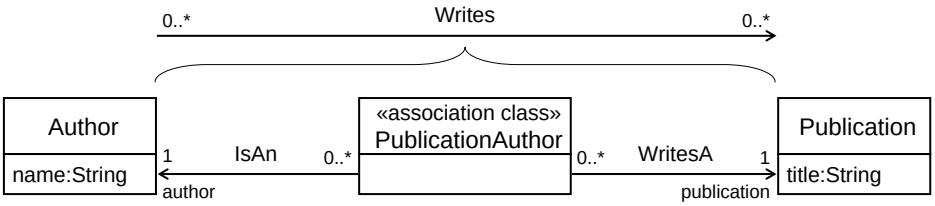


Figure 3.9.: Association class that realizes a many-to-many relationship.

Thus, the relationship *Writes* could be modeled as *association class PublicationAuthor* (cf. Fig. 3.9) that utilizes the two many-to-one associations *IsAn* and *WritesA*. This way one is able to relate also many authors with one publication in the class diagram system modeled by LM_{CD}.

Note that it is also possible to specify a mapping that directly supports many-to-many associations in LM_{CD}. Therefore, the syntax definition of LM_{CD} (cf. Fig. 3.2) has to be extended by multiplicity information (whether the association is one-to-one, one-to-many, many-to-one, or many-to-many) which is then taken into account by the mapping specification. But, this is only an optional requirement for a CDDS mapping specification which will not be implemented in the CDDS mapping specification discussed in this thesis.

There is one specialty when translating a schema (backwards) into a class diagram based on the mapping definitions. Columns are either data columns that are used to store values of primitive type or key columns (i.e., primary key column or foreign key column) that store identifier values. According to the given constraints in LM_{DS} all columns of a table are ordered. That is, foreign key columns—which represent relations in the target domain—are also ordered. Therefore, the target domain contains (ordering) information that is not intended in the source domain. This either has to be taken into account when specifying the mapping or this additional information is lost. In our mapping specification this ordering information is intentionally lost.

Now, we have argued which elements of LM_{CD} and LM_{DS} should be mapped onto each other. Based on the mapping description we expect that translators produce a corresponding representation of models of one language (either object or data domain) in the related language (either data or object domain). That is, given package 3.8(d) as input to a "CD to DS" translator (i.e., forward translator) we assume that schema 3.8(e) or an equivalent schema, where the ordering of columns in table $t1$ slightly differs, is produced as output. Likewise, a "DS to CD" translator (i.e., backward translator) should produce package 3.8(d) given schema 3.8(e) as input. In this special case the translator will always reproduce package 3.8(d) because there exists no equivalent package—even though with an alternate sequence of patterns.

So far, we have not yet discussed *how* instances of LM_{CD} and LM_{DS} (i.e., class diagram and database schemata models; cf. Fig. 3.8) are mapped onto each other. An intuitive approach would be to somehow relate the patterns depicted in Figs. 3.3, 3.4, 3.5, and 3.6. For this purpose we will use the triple graph grammar formalism which is explained in Chap. 4.

3.4. Similarities in Natural and Formal Language Translation

Now that we have discussed an example of bidirectional natural language translation and bidirectional formal language translation in the preceding section we will abstract from these examples and examine the similarities of both activities in general.

The need for translators arises in situations where one stakeholder (human or computer system) does not speak one language (well) and demands that documents/models expressed in this foreign language are translated to a familiar language. If two stakeholders work on the same document (or related ones) but do not speak the same language, bidirectional translators are requested. Transfered to the CAB example where Chris and Alex work on the same content, Chris adds a new section to an already existing chapter to her version of the book (i.e., the new section is written in german). Alex who likes to integrate this new content in his english version of the book has to somehow translate the content. He utilizes Binaltas likewise to Chris who had utilized it when translating Alex' original version earlier. In CDDS, a software architect would work in cooperation with a database expert in order to design a maintainable high-performance system by utilizing CD<>DS to translate ideas from the object-oriented domain to the data domain and vice versa. Moreover, even experts in many languages demand for translators because some languages are more appropriate in some cases than others to express different aspects. That is, experts often change their point of view by changing to another more appropriate language. Therefore, such experts demand for translators that automatically represent a model as a corresponding model of another language.

Reviewing Fig. 3.1 we see that the situation in the CD<>DS example is quite similar to the CAB example. A main difference is that during formal language translation a computer based translation system should take the role of the translator—instead of a human being. Further, the translation system should not interact with humans that often during the translation process. In addition, we could add a new enumeration literal *CorrespondsWith* to the enumeration *RelationType* of the domain model depicted in Fig. 3.1. A model relation of this type could then be added as metainformation to relate two corresponding models that were produced by a translation system. However, such a relation is too coarse-grained because it just states that two models are related but not which elements of the model correspond to each other. Our approach discussed in Sect. 3.6 supports managing relationships of mapped elements—by utilizing traceability links—on a fine-grained level.

When specifying a mapping between two languages, experts of both languages (i.e., users of the language and the ones that have knowledge about the language specification) should be involved that are able to interpret models of the language [KRS08]. Moreover, language translation experts should be involved in the specification process. All involved experts will communicate with each other while specifying the mapping. Note that there might exist different *mapping strategies* which have their advantages and disadvantages (cf. Sect. 3.3.5). Like in CDDS, where different solutions exist for mapping the class hierarchy onto tables.

In both cases—natural and formal translation—the participating languages provide a certain set of features. Some of these features are shared by both languages like *typing* and *relating*. A model of one language that shares some features with another language can be

(partially) translated into a model of the other language. This is true for natural languages and formal languages. In the CAB example, a book about a subject written in english is translateable into a book about the same subject written in german. In CDDS an object-oriented language is related with a data-oriented language, i.e., class diagram models are related with database schemata models (cf. Figs. 3.8 (d) and (e)). Mapping features *one to one* is far more easy than mapping features *not one to one* (cf. Sect. 3.3.5). In CDDS, mappings of features that are (almost) one to one are the mappings *Class<>Table* and *Attr<>Column*. In the CAB example, a feature which exists in both languages is the type *pitch*. However, the names of the pitches are slightly different which has to be taken into account in the mapping specification. "B" is related with "H" and "B_b" is related with "B" . In CDDS the name of a relation is mapped differently (cf. Fig. 3.5).

Due to features not provided by one of the languages information is lost in some cases. In CDDS, the ordering information of key columns will be lost when translating a database schema into a class diagram. In natural language this loss of information occurs, e.g., when mapping the german formal adressing with "Sie" to the english "you". Mapping "you" backwards to the german language may result in "Sie" or in "Du" depending on the context and information that is perhaps not present in the mapping system (i.e., whether the correspondents are on a first-name basis).

An important issue in translation processes is *encoding of information*. For example, if a language does not support a feature of the opposite language, the (additional) information type of the not-supported feature may be related with a supported feature or a combination of supported features. In CDDS, the inheritance feature of CD is not basically supported by DS. Therefore, the inheritance structure of class diagrams is encoded using the foreign key construct in the corresponding database schema. Another example where features are not mapped one to one in CDDS is the mapping of relationships (cf. Fig. 3.5). There, associations are related with a combination of yet unused elements (i.e., foreign keys) and elements still in use by the properties feature (i.e., columns). Encoding of additional information (e.g., by using generic elements) might also lead to reduced functionality due to required reserved words or reserved structures. For example, CDDS utilizes operation "concat" in pattern "relateTableWithTable". This operation internally uses the character '@' to mark concatenation points. This prohibits using the character '@' in names of tables and foreign key columns if the inverse operations "splitGetFirst" and "splitGetSecond" should be able to extract the tokens from the concatenated string. Otherwise the split operations might split at a wrong "concatenation point"—that is part of a token. We will see in Sect. 5.2.3 that these inverse operations are required and that the character '@' must not be used for names of classes and associations too in order to achieve "real" bidirectional translations.

Concerning the two issues—mapping features and encoding of information—discussed above, [KWB03] states (a):

> *[...] complete bidirectional transformations between models are*[14] *only possible if the expressive power of the source and target modeling languages is identical.*

and (b)[15]:

[14] In [KWB03] it is "is" instead of "are".

[15] We slightly rephrased (b) to source/target-independent terms without destroying the original statement.

> *If additional information is added to one language, or if there is information in one language that is not mapped to the other language, bidirectionality is impossible to achieve.*

The pitch example also shows that mappings may change over time because one of the languages evolves. In this case the mappings have to be adjusted. Likewise, the CAB example shows that changes might occur in mapped models. In several real world scenarios, changes occur in both models which demands for bidirectional translators instead of only providing unidirectional translators [CFH+09].

Once a model has been translated, the corresponding model elements should be traceable [CH03]. In the CAB example, Binaltas keeps track of related words in the original publication and the translated version. This allows translators to provide an additional *incremental mode* (besides the normal *batch mode*) which is able to *synchronize* source and target model if they change. Due to performance issues this mode typically relies on metainformation that has been added during previous translation processes, e.g., correspondence graphs (cf. [Sch95]) or traceability links (cf. [GW09]). In general a translator doing incremental updates has higher performance compared to the batch mode which throws away the outdated model and retranslates the modified model. Performance is especially crucial when synchronizing large models. In addition, the incremental mode should preserve additional information present in the to-be-adjusted output model.

3.5. Challenges Realizing Bidirectional Translators

We will now summarize the challenges that have to be faced when providing support for building bidirectional (model-to-model) translators and state these challenges as requirements for bidirectional translators. Challenges that arise when building bidirectional translators based on the triple graph grammar approach have been stated, e.g., in [Sch94], [KKS07], [SK08], and [KLKS10]. The discussion in the previous sections showed that these challenges are transferrable to the more general situation of building bidirectional translators based on any translation language. Moreover, additional requirements, which are discussed in [CFH+09] and [Ste10], will be added to the set of requirements.

First, we will state requirements that have to be met by languages that are used to specify bidirectional translators (bx-language):

bxL1 A bx-language should support a *general approach* for mapping various kinds of DSLs, i.e, the bx-language should be a *geneneral-purpose bidirectional model transformation language* (GPbxL).

bxL2 A bx-language must have *precisely defined semantics*.

bxL3 A bx-language should be *expressive* in order to be "useful" in practice. Common mapping scenarios must be supported.

bxL4 A bx-language should support *non-bijective bidirectional translations* due to cases where both input and output models contain almost (but not quite) the same information. In

such cases, many models exist in the output domain of a translator that might be related to the input model.

bxL5 A bx-language should provide mechanisms that allow to *reuse* parts of model translation specifications.

Note that we discussed *bxL4* by example in Sect. 3.3.5. There, a "CD to DS" translator might produce equivalent output models depending on its related mapping specification.

Second, we state requirements that have to be met at runtime of bidirectional translators (bx-translators) rather than at specification time of a mapping:

bxT1 A translation process must always *terminate*. A very basic and intuitive property of translators.

bxT2 Translators must be *correct* (cf. Sect. 4.7) with respect to their mapping specification. This definitional property is also called "consistency" in [SK08] and [KLKS10]. However, the term "correctness" is more appropriate and used e.g., in [Sch94], [EHS09], and [Ste10].

bxT3 Translators should be *complete* (cf. Sect. 4.7) with respect to their mapping specification.

bxT4 Translators should be executable *efficiently* in order to be useful.

bxT5 Translators must not modify two corresponding models if they are already consistent according to the mapping specification. In [Ste10] this definitional property is called *hippocraticness*.

bxT6 Translators should support the quality property *incrementality*, i.e., allow to perform incremental change propagations in order to synchronize corresponding models.

bxT7 Translators should handle *additional information* present in a model of one domain in such a way that it is not deleted if the user wants to keep this additional information. This requirement is related to the incrementality property.

This thesis does not address the incrementality property, i.e., construction of incrementally working translators. We leave it up to future work to transfer the results of this thesis to sophisticated incremental approaches. We also omit the optional definitional property *undoability* which is related to incrementality and described in [Ste10] as the following process of operations. Let m and n be a consistent pair of corresponding models. Model m is modified—resulting in $m0$. The changes are propagated to model n such that it is replaced by $n0$. Immediately, without making any other changes to either model, the model $m0$ is reverted to the original version m. The changes are propagated to model $n0$. In this case, a user expects that the effect of the modification has been completely undone, i.e., $n0$ is returned to its original state n. This property is omitted in this thesis because the author of this thesis is of the opinion that this property is a rather technical property of a translation system and can, e.g., be realized by versioning control mechanisms.

As stated in previous work (cf. [SK08, KLKS10]) translators should be *efficient* (i.e., have polynomial space and time complexity) and *compatible* with respect to their mapping specification, i.e., be correct and complete (cf. Sect. 4.7). In addition, the language used to specify the mapping must be *expressive* enough. In this thesis we will concentrate especially on the expressiveness property of the bx-language and on the properties efficiency, correctness, and completeness of translators that work in batch mode. We will discuss these properties in more detail in the next chapters.

3.6. Model-Driven Language Integration with TiE

In Sect. 2.5 we have discussed that *bidirectional transformations* are used for maintaining the consistency of two related sources of information. According to [KWB03] and [CFH+09] bidirectional transformations can be achieved in two ways. On the one hand two unidirectional transformation definitions are specified such that they are compatible to each other (i.e., "somehow" inverse). On the other hand both transformations are performed according to one bidirectional transformation definition. In the following we will concentrate on the second approach. We consequently follow the model- and language-driven approach discussed so far and will use a formal model transformation language specialized in the *language mapping and translation domain* to describe bidirectional mappings.

The approach to realize model-based bidirectional translators used in this thesis is based on the tool integration environment *TiE* presented in [KRS09] and on the (meta-)model-driven development (MMDD) process presented in [KRS08]. TiE utilizes the meta-CASE tool MOFLON [AKRS06] which provides an implementation of the MOFLON specification language (MOSL) [Ame09, Kön09]. This specification language composes the languages MOF, OCL, SDM, and triple graph grammars (TGGs) and allows to create mapping specifications in a model-driven process.

The architecture of the TiE approach is depicted in Fig. 3.10. The left-hand side of the figure depicts the relationship of the components that are produced in the analysis and design activity of the MMDD process. The right-hand side depicts the relationship of the components that are produced for the implementation activity of the MMDD process. The latter components are executed at runtime of a tool integration scenario.

The *requirements of the tool integration project* are the first artifact produced in the MMDD process. This is a vital artifact because all other components depend on it. For a detailed explanation of this activity, which is aligned with common software development processes, we refer to [KRS08]. Taking the integration requirements into account, the mapping is specified as a TGG (denoted as hexagonal element in Fig. 3.10) that consists of a TGG schema and a set of TGG productions. The TGG relates corresponding elements of two languages A and B. Supported languages are DSLs that are, e.g., implemented in a certain tool (i.e., software system). In general, our approach allows to map pairs of DSLs that share some features onto each other (cf. requirement *bxL1*). The TGG schema defines the link types of the correspondence links—or traceability links—that are established between corresponding elements during runtime. TGG productions specify how elements in both related languages and traceability links are created simultaneously.

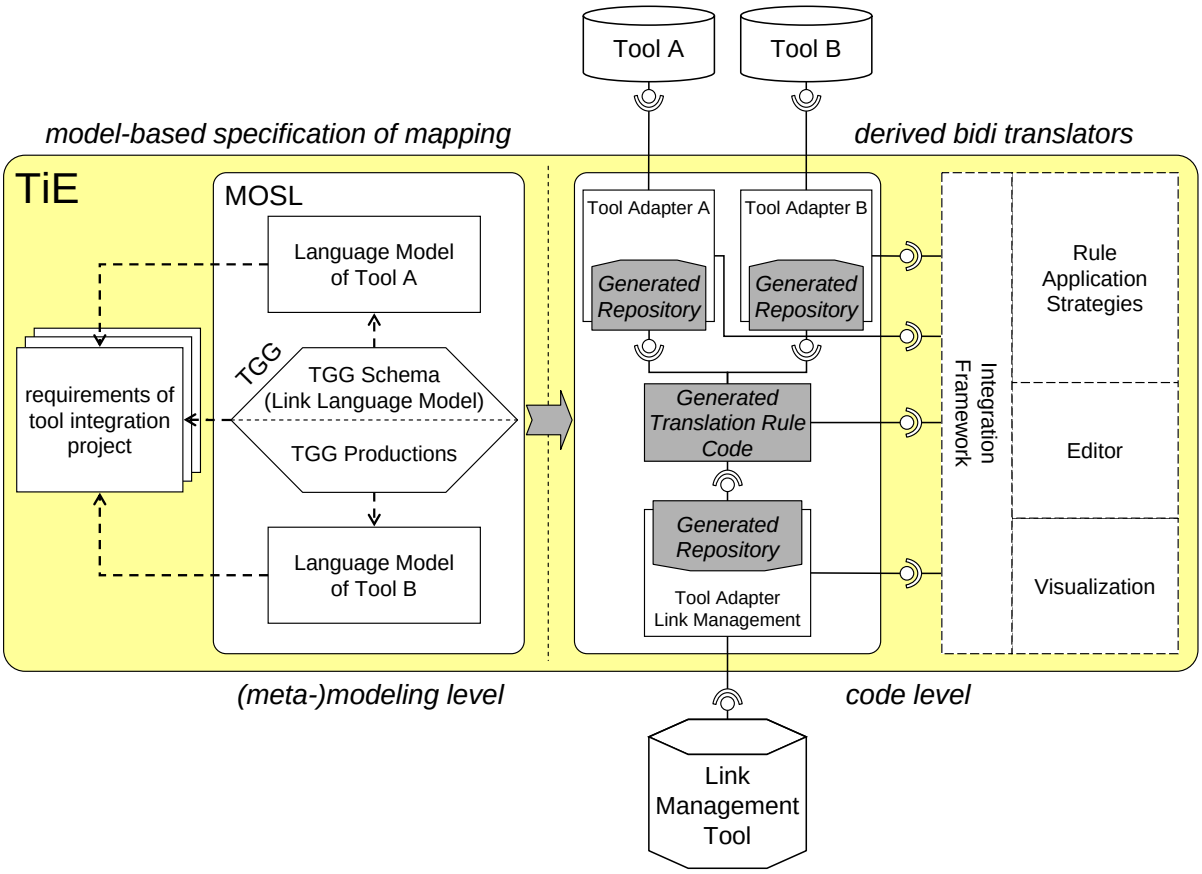


Figure 3.10.: Architecture of the tool integration environment TiE (adapted from [KRS09]).

After the mapping specification is completed, MOFLON generates so-called *repositories*: components that correspond to the language models and mapping specification respectively. Each repository contains a standard implementation of its related language model. This includes data storage and data manipulation functions. In addition, well-defined interfaces are provided that allow to access the language's elements and its instances programmatically. Moreover, a repository has the capability to serialize its (meta)data using XMI in order to exchange data with a related tool. The *translation rule* component, which is derived from the set of TGG productions, contains modification and checking routines that operate on runtime instances of tool models and traceability links. This component provides the basic operations required to achieve forward and backward translation, checking for consistency of related tool elements, and performing incremental updates. In order to realize these operations the translation rules require access to the interfaces of all generated repositories.

The implementation of a generated repository is extended to a so-called *tool adapter*. Therefore, the generated implementation is adjusted to use the interface provided by the adapted tool. The tool adapter uses for example the API of the tool and forwards access calls "online" to the tool via this API. In this case, data management takes place in the tool—not in the tool adapter. But, a tool adapter need not to directly access the tool it is responsible for and the implementation of the generated repository may be used as is. In this case, tool data is mirrored and managed by the generated repository and exchanged "offline" with the tool, e.g., via the XMI mechanism. The interface provided by a tool adapter must include the interface definition provided by its repository.

Our approach allows to reuse tool adapters and language models for other integration scenarios. That is, if a mapping is going to be defined between a tool A and a tool C and tool A had been integrated with another tool B earlier, the language model of A and its tool adapter could be reused. Therefore, creating tool adapters and language models is a one-time effort.

The mapping specification component, i.e., TGG, requires a rather high-level view on a system. Therefore, a language model should look, e.g., like depicted in Fig. 3.2 using the modeling-concept of associations to relate elements rather than modeling a proprietary ID-based relation mechanism (which might be implemented in a tool). Therefore, language models have to be designed using common model-driven design principles in order to be conveniently usable by a mapping specification. Consequently, a tool adapter allows to abstract from a tool's data structure implementation. That is, the language model of a tool need not to reflect the tools data structure one-to-one. This allows integrating tools that have a rather low-level implementation—which does not reflect the concepts used in a higher-level modeling language—and that do not support an XMI mechanism.

Finally, Fig. 3.10 depicts the *integration framework* of our approach that utilizes the tool adapter components and the translation rule component. In conjunction with certain *rule application strategies* (i.e., algorithms that perform various translation operations) the framework supports various bidirectional translation scenarios. The framework controls certain translation processes such as forward and backward translation. In addition, the framework supports visualization of tool elements and traceability links, as well as basic editing operations on the model elements.

So far, we have discussed the concept of bidirectional language translation and the basic

parts utilized by our approach to model-driven specification of bidirectional translators. In the chapters to come we will describe the parts used to realize bidirectional translators in detail: theory and application of triple graph grammars, derivation of translation rules, and implementation of rule application strategies. Moreover, we will keep in mind the requirements for bidirectional translators stated in Sect. 3.5.

4. Triple Graph Grammars

> *There is some sort of abstract "conceptual skeleton" which must be lifted out of the levels before you can carry out a meaningful comparison of two programs in different computer languages, [...]*
>
> [Hofstadter [Hof99, p. 381]]

In the previous chapters we motivated the need for bidirectional language translation and discussed our model-driven language integration approach TiE and the fundamental concepts required for its realization. TiE utilizes triple graph grammars to relate corresponding elements of two languages by so-called correspondence links. Now, we will discuss how to specify bidirectional mappings with *triple graph grammars* (*TGGs*) in order to realize bidirectional language translators. Section 4.1 gives a general overview to TGGs. Then, the TGG formalism is explained by building a triple graph grammar specification—called TGG_{CDDS}—that maps class diagrams and database schemata onto each other. This TGG will implement the mapping requirements for the language models LM_{CD} and LM_{DS} collected in Sect. 3.3.5. The syntax specification of TGGs is discussed in Sect. 4.2 whereas TGG productions that evolve graph triples are discussed in Sect. 4.3. The set of TGG productions for TGG_{CDDS} is then presented in Sect. 4.4. Next, Sect. 4.5 explains the simultaneous evolution of graph triples by example. Section 4.6 shortly discusses the most vital feature of triple graph grammars: the ability to derive forward and backward translation rules from one TGG specification. Together with sophisticated rule application strategies, these translation rules form bidirectional language translators. Finally, Sect. 4.7 discusses the challenges arising in the TGG approach: the TGG language must be expressive enough in order to support common translation scenarios but the correctness, completeness, and efficiency properties of derived translators need to be ensured.

4.1. Overview

Triple graph grammars are a formalism that allows for the specification of correspondence relationships between two languages of graphs. The concept of *triple graph grammars* was introduced in [Sch94, Sch95][1]. TGGs have been invented to support translation of documents based on related graph-like data structures. As discussed in Sect. 2.8 this also includes documents that can be represented as model. The main feature of the *TGG language* is the ability to derive bidirectional working translators from a *TGG specification*. A TGG specification is

[1] In contrast to the short version of triple graph grammars published in [Sch95], the technical report [Sch94] contains an algorithm that is used to realize translators based on triple graph grammars.

typically created in a model-driven way and fits well in the MDE world (cf. Sect. 2.4) and the relational part of OMG's model transformation standard QVT (cf. [GK07, Kön09, GK10]). TGGs are grammars that generate graph triples by applying productions of the grammar to an input graph triple (axiom). Therefore, TGG productions specify the simultaneous evolution of triples of graphs. This allows to derive unidirectional forward and backward translation rules from one TGG production. In addition, TGGs allow to check consistency of related documents and to efficiently restore consistency if changes occur in a document.

The three graphs of a *graph triple* GT are often called *source graph* G_S and *target graph* G_T—both representing the elements of the related languages—and *correspondence graph* G_C. The elements of the correspondence graph are called *correspondence links* and contain additional information about the translation process. This allows for the realization of incremental updates that are required if changes occur in the source or target model and the corresponding model needs to be synchronized. The correspondence links serve as *traceability links* because they allow to trace elements in both languages that correspond to each other. Correspondence links are elements of a language, too—the *correspondence language* or *link language* that relates elements of the source and target language. Throughout this contribution we will use the terms "correspondence link" and "*TGG links*" interchangeably. It is important to notice that TGG links are nodes in terms of the graph domain—and objects in terms of the modeling domain—but that TGG links are neither edges nor "ordinary" links.

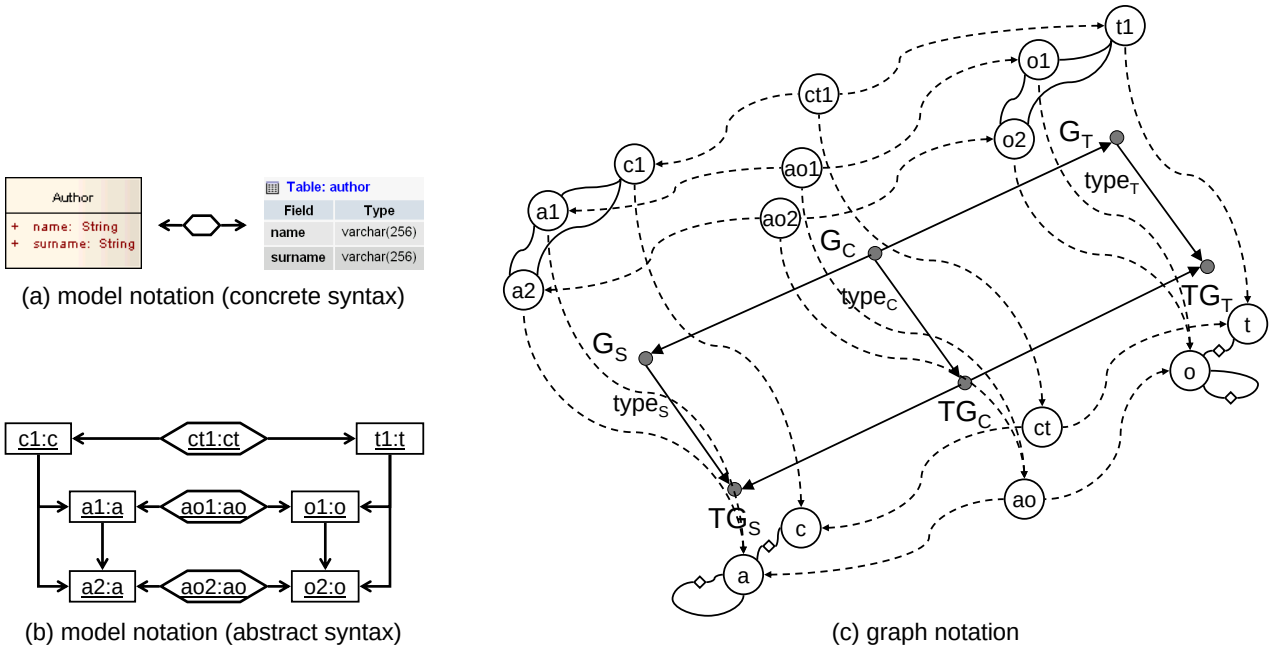


Figure 4.1.: A graph triple: model notations and graph notation.

Fig. 4.1 (a) depicts three models. The source model—an instance of LM_{CD}—consists of a class *Author* which contains two attributes. The target model—an instance of LM_{DS}—consists of a table *author* which contains two columns. The link model—denoted as hexagonal element—contains correspondence links that relate corresponding elements of source and

target model. Figure 4.1 (b) depicts the three models in abstract syntax. Note that information has been omitted in Fig. 4.1 (b): class, attribute, table, and column names as well as attribute and column types (i.e., *String* and *varchar*) are not depicted. Moreover, the names of the classifiers of the source and target language **c**lass, **a**ttribute, **t**able, and c**o**lumn have been abbreviated. Figure 4.1 (b) depicts TGG links on a more fine-grained level and allows to trace instances of source and target language that correspond to each other. Figure 4.1 (c) represents the triple of models depicted in Fig. 4.1 (b) as graph triple. The inner elements $G_S \xrightarrow{type_S} TG_S$, $G_C \xrightarrow{type_C} TG_C$, and $G_T \xrightarrow{type_T} TG_T$ depicted in Fig. 4.1 (c) are a schematic representation of the typed graphs of source, correspondence, and target domain (cf. also right-hand side of Fig. 2.17). Such schematic notations will be used in formal representations of triple graph grammars throughout this thesis. Note that morphism arrows from links in source and target graph to their classifying associations have been omitted in Fig. 4.1 (c).

The original TGG approach (cf. [Sch94]) is based on definitions of simple graphs. In this thesis we will use the more elaborate definitions of constrained typed graphs given in Sect. 2.6 for the definitions of an extended TGG approach. Definition 11 describes the conditions that must be satisfied by *constrained typed graph triples*. These conditions are analogous to the conditions that must be satisfied by *constrained typed graphs* (cf. Def. 5). Each graph of a graph triple GT is in the set of graphs of a particular language, i.e., conforms to a graph schema defined by a certain (constrained) type graph, e.g., LM_{CD} and LM_{DS}. Two morphisms h_S and h_T relate elements of the correspondence graph G_C with elements of the source graph G_S and target graph G_T respectively. In conjunction, these two morphisms are an essential part of triple graph grammars as they allow to trace corresponding elements in a graph triple. The morphisms allow to identify corresponding elements, e.g., $c1$ and $t1$ which are elements of the graph triple depicted in Fig. 4.1 (c). These two elements of source and target domain are related via the same correspondence node $ct1$, i.e., $c1 = h_S(ct1) \wedge h_T(ct1) = t1$; also denoted as $c1 \leftarrow ct1 \rightarrow t1$ or $c1 \leftrightarrow t1$. In the following we assume that all graphs with suffix "S", "C", and "T" have type graphs TG_S, TG_C, and TG_T respectively.

Definition 11. *Graph Triple.*
Let G_S, G_C, and G_T be three graphs with morphisms $h_S : G_C \rightarrow G_S$ and $h_T : G_C \rightarrow G_T$ that represent m-to-n relationships between G_S and G_T via G_C in the following way:
$x_S \in G_S$ *is related to* $x_T \in G_T \ :\Leftrightarrow \exists x_C \in G_C$: $x_S = h_S(x_C) \wedge h_T(x_C) = x_T$.
Then $GT := (G_S \xleftarrow{h_S} G_C \xrightarrow{h_T} G_T)$ *is a* graph triple.

Now, we will lift the definition of typed graphs and type preserving graph morphisms (cf. Def. 4) to *constrained typed graph triples* and *type preserving graph triple morphisms*. Constraints for correspondence graphs are disregarded in Def. 12, but can be added easily if needed.

Definition 12. *Constrained Typed Graph Triples and Type Preserving Graph Triple Morphisms.*
Let $\mathcal{L}(TG_S, \mathcal{C}_S)$ and $\mathcal{L}(TG_T, \mathcal{C}_T)$ be languages of constrained typed graphs of source and target domain of a graph triple GT, whereas $\mathcal{L}(TG_C)$ defines a language of correspondence graphs of type TG_C that relate pairs of source and target graphs. GT is a properly typed *graph triple iff*

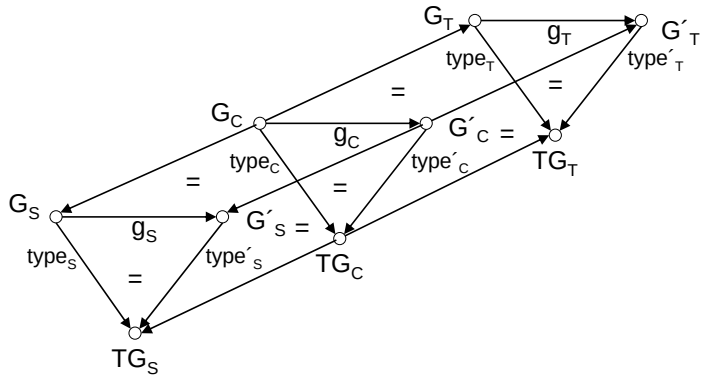


Figure 4.2.: Type preserving graph triple morphism (g_S, g_C, g_T).

(1) $G_S \in \mathcal{L}(TG_S, \mathcal{C}_S)$*, (2)* $G_C \in \mathcal{L}(TG_C)$*, (3)* $G_T \in \mathcal{L}(TG_T, \mathcal{C}_T)$*.*
A type graph triple $TGT := (TG_S \overset{h_S}{\leftarrow} TG_C \overset{h_T}{\rightarrow} TG_T)$ *is a distinguished graph triple.* TGT *together with morphisms* $type_S : G_S \rightarrow TG_S$*,* $type_C : G_C \rightarrow TG_C$*,* $type_T : G_T \rightarrow TG_T$ *is called* type *of* GT*. A* graph triple morphism (g_S, g_C, g_T) *with* $g_S : G_S \rightarrow G'_S$*,* $g_C : G_C \rightarrow G'_C$*,* $g_T : G_T \rightarrow G'_T$ *is* type preserving *iff the so-called "toblerone" diagram (cf. Fig. 4.2) commutes.* $\mathcal{L}(TGT)$ *is the set of all graph triples of type* TGT*.*

Consequently, the graph triple depicted in Fig. 4.1 (c) is not properly typed (i.e., not *well-formed* or *invalid* according to the terms introduced in Sect. 2.2.4), relating to LM_{DS}. This is due to the fact that G_T does not contain a primary key and, therefore, violates a multiplicity constraint of its graph schema because each table of LM_{DS} must have at least one primary key (cf. Fig. 3.2). But the graph triple depicted in Fig. 4.1 (c) can be easily modified such that it becomes a properly typed graph triple relating to LM_{CD} and LM_{DS} by adding a primary key and a primary key column to G_T.

Type preserving graph triple morphisms will be used in Def. 17 when defining graph triple rewriting. Morphisms that are *type preserving* guarantee that elements do not change their types, e.g., if they are rewritten by graph productions. A graph triple $GT \in \mathcal{L}(TGT)$ is always *syntactically correct* but it need not to be *semantically correct* because it may violate additional well-formedness rules, i.e., additional graph constraints (cf. graph triple depicted in Fig. 4.1 (c)).

4.2. TGG Schema

TGGs consist of a *TGG schema* which describes the structural dependencies between the elements of the two related languages and the correspondence language. The elements of both languages are related via so called *correspondence link types* which are shortly called *link types* and are located in the *link domain*. The notation of link types is similar to the notation of classes except that link types are denoted with a hexagonal border instead of a rectangular border. TGG links are instances of link types.

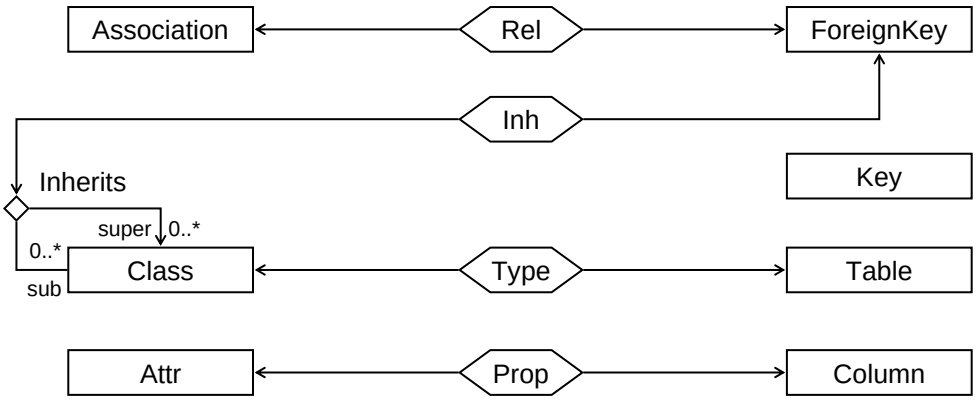


Figure 4.3.: TGG schema of TGG_{CDDS} that relates class diagrams and database schemata.

Figure 4.3 depicts the correspondence language of the TGG schema of TGG_{CDDS} and the elements of the source and target language, i.e., class diagrams and database schemata, that are related by the correspondence language. The complete sets of elements of the source and target language are depicted in Fig. 3.2. These languages and their constraints have been discussed in Sect. 3.3 and will be reused in the triple graph grammar TGG_{CDDS}. The link type *Type* of TGG_{CDDS} relates classes and tables, whereas the link type *Prop* relates attributes and columns. The elements *Association* and *ForeignKey* that are used to realize the relationship feature in both languages are related via the correspondence link type *Rel*. Finally, link type *Inh* serves as classifier for TGG links that relate inheritance information present in both languages. Class diagrams decode inheritance relationships with instances of the association *Inherits*, whereas in database schemata, the foreign key concept is used to decode inheritance relationships. Therefore, link type *Inh* relates association *Inherits* and class *ForeignKey*.

4.3. TGG Productions

In addition to the TGG schema, a set of *TGG productions*[2] is specified for each triple graph grammar. In conjunction with the TGG schema, TGG productions define a language $\mathcal{L}(TGG)$ of consistent graph triples, i.e., they describe how related triples of graphs may evolve simultaneously and how the elements of the corresponding languages relate to each other. Likewise to graph productions (cf. Sect. 2.7), each TGG production consists of a left-hand side L and a right-hand side R. The left-hand side is a graph pattern that looks for a corresponding match (*redex*) in a graph triple. If applied to a redex, a TGG production adds a copy of the elements of its right-hand side that are not already part of the left-hand side to the regarded graph triple.

We have already learned in Sects. 2.8.3 and 3.3.3 that negative application conditions (*NACs*) can be added to graph productions. NACs can be used, e.g., to avoid creation of

[2]TGG productions are often called *TGG rules* in other publications. However, we stick to the term "TGG production" to avoid clashes with *translation rules* that are derived from a TGG production.

graphs that violate constraints defined in the graph schema. We will discuss the problems that arise when adding the concepts of NACs to the TGG formalism in Chap. 5 in detail. For now, we will assume that NACs may be added to TGG productions.

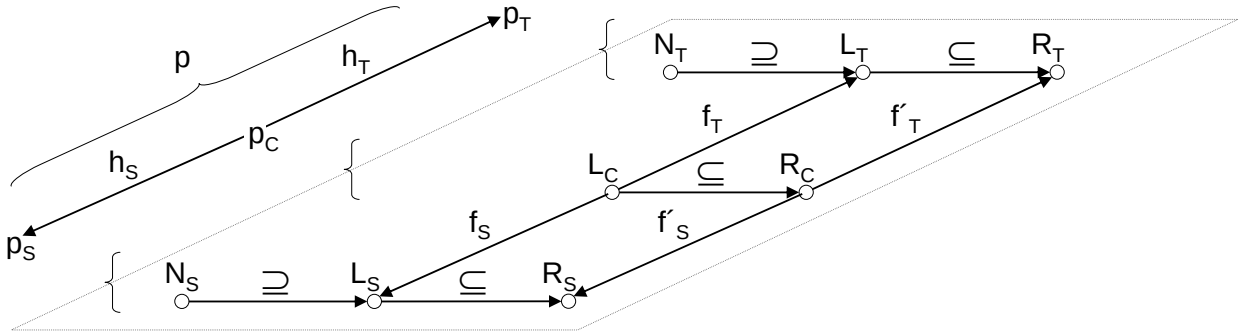


Figure 4.4.: Schematic view of a TGG production.

Figure 4.4 schematically depicts a TGG production $p := (p_S \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$ that supports NACs. The TGG production consists of three production components p_S, p_C, and p_T that rewrite the source, correspondence, and target graph respectively. The source component $p_S := (L_S, R_S, \mathcal{N}_S)$ and the target component $p_T := (L_T, R_T, \mathcal{N}_T)$ of a TGG production are graph productions with sets of NACs. The correspondence component p_C is a simple graph production (L_C, R_C) without sets of NACs. The left-hand side $L := (L_S \overset{f_S}{\leftarrow} L_C \overset{f_T}{\rightarrow} L_T)$ of p consists of the left-hand sides of its components. The right-hand side $R := (R_S \overset{f'_S}{\leftarrow} R_C \overset{f'_T}{\rightarrow} R_T)$ consists of the right-hand sides of its components.

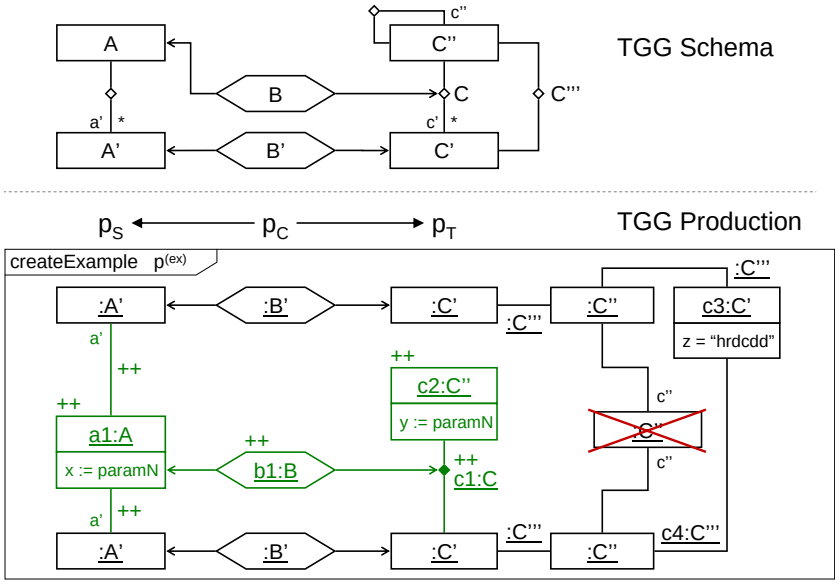


Figure 4.5.: Abstract example of a TGG production.

Figure 4.5 depicts an abstract example of a TGG production $p^{(ex)}$ based on a TGG schema depicted in the upper part of the figure. This example reflects all relevant production elements

that are used throughout this thesis. Likewise to SDM patterns, we use a shorthand notation for TGG productions. Instead of showing both left-hand side and right-hand side as two separate parts of a production, both sides are merged (cf. Sect. 2.8.3). Moreover, the depicted elements are objects and links belonging to the domain of models instead of nodes and edges belonging to the domain of graphs (cf. Sect. 2.8 for a mapping of models to graphs; cf. Fig. 4.1 for a comparison of graph triples in model and graph notation). The hexagonal elements are TGG links that belong to the correspondence component p_C of a production. The notation of TGG links is similar to the notation of objects except that TGG links are denoted with a hexagonal border instead of a rectangular border to better distinguish elements contained in the different components of a production.

The elements contained in the left- and right-hand side of a TGG production $L \cap R$ are *context elements* that define a pattern that matches a redex in a host graph triple to which the production is then applied. Context elements are denoted as black elements without any additional markup. In the example, the expression z = *"hrdcdd"* denotes a context element which matches a slot (cf. Fig. 2.23) in the host graph that is owned by the matched object $c3$. The slot must be an instance of property z and must have the value *"hrdcdd"*.

The elements contained in the right-hand side only $R \backslash L$ are created during the application of a TGG production to a redex. They are denoted as green elements with an additional ++ markup. An element that is created by a TGG production is called *primary element*—or more specific *primary node* or *primary edge*. In our approach, each source and target production component contains at most one primary element and the total quantity of primary elements in source and target components must not be zero. The correspondence component must contain exactly one primary element—a primary TGG link. Primary elements of source and target domain ($a1$ and $c1$) are attached to the primary TGG link ($b1$). In general, TGG productions may create additional *secondary elements*[3] (e.g., $c2$) that are directly or transitively connected with the primary element of their production component. Edges that are incident to a primary node are always in $R \backslash L$, i.e., they are always created by a production. Such edges are included in the set of secondary elements. Nodes that are incident to a primary edge need not to be created by a production. But if they are created by a production then they are secondary.

In our approach, all primary elements of a TGG production are primary nodes. This is due to the fact that links—like $c1$—consist of one *link node*, two *slot nodes*, and four edges according to our mapping of models to graphs (cf. Fig. 2.24 and Def. 10). Thus, the term "*primary link*" refers to the *link node*, which is primary, and the slot nodes and edges, which are secondary. The term "*secondary link*" refers to all elements of the link as secondary elements.

A TGG link—like $b1$—connects either to an object or to a link. If it connects to a link (e.g., $c1$), it relates to the link's *link node*—which in the case of $c1$ is an instance of association C's *association node*. In graph terms, the node of TGG link $b1$ depicted in Fig. 4.5 is related via morphism h_T to the *link node* of link $c1$. Likewise to the concrete syntax notation for graphs, a link is denoted as solid diamond in this case.

Our approach also supports *TGG parameters* (cf. [Kön09]). These parameters are given to a TGG production during production application. A slot that is created by a TGG production and *bound* to a TGG parameter will use the value of the TGG parameter as its own value. In

[3] Secondary elements are also called *non-primary elements*.

Fig. 4.5, the value of TGG parameter *paramN* will be used as value for the slots of properties x and y that are owned by the newly created objects $a1$ and $c2$ respectively.

TGG productions are usually *monotonic*, i.e., they do not delete any graph elements (cf. Def. 7 in Sect. 2.7). Therefore, the set of elements $L \backslash R$ that are contained in the left-hand side only must be empty. Consequently, all left-hand side graphs of TGG production components are subgraphs of their right-hand side graphs. According to [Sch94], monotonicity is no disadvantage because TGGs are not intended to model editing processes on related graphs—that is, they are not intended to realize triple graph rewriting/replacement systems—but are a generative description of graph languages and their relationships and are used to derive bidirectional language translators. Such translators get either a graph of the source or target domain as input and have to translate this graph into a graph of the corresponding domain. Therefore, translators have to realize an efficiently working graph parser in order to reconstruct a proper sequence of TGG production applications whose input components (either p_S or p_T) have created the given input graph. Monotonicity considerably simplifies development of bidirectional working forward and backward translators based on TGGs because an input graph directly contains all information about its derivation history.

4.4. Productions of TGG_{CDDS}

Based on the TGG schema depicted in Figs. 4.3 and 3.2 and the discussion in Sect. 3.3 we will now start building the set of TGG productions for TGG_{CDDS} that specifies the mapping between class diagrams and database schemata. In Sect. 3.3.5 we collected the requirements of a mapping between CD and DS and mentioned to map CD and DS by somehow using the graph productions depicted in Figs. 3.3, 3.4, 3.5, and 3.6 (cf. Sect. 3.3.3 for a detailed discussion of these patterns). In the following we will see that most of these patterns can be used as source and target production components when constructing TGG productions of TGG_{CDDS}. This is due to the fact that modeling a TGG production is similar to modeling a graph production. The difference is that a graph production is "one-dimensional", whereas a TGG production is "three-dimensional", i.e., consists of three graph production components (cf. Sect. 4.3).

The TGG productions of TGG_{CDDS} are depicted in Fig. 4.6. Production "createType" $p^{(t)}$ creates elements that realize the concept of *typing* in CD and DS and relates the corresponding elements via a TGG link. Therefore, a new persistent class $c1$ is created in the source domain and a new table $t1$ in the target domain. A newly created TGG link $l1$ of type *Type* relates $c1$ and $t1$. Note that a parameter n of type *String* has to be passed to the TGG production which is assigned to the *name* of class $c1$ and table $t1$. Elements $c1$, $l1$, and $t1$ are *primary nodes*. The target component contains additional *secondary objects*, a primary key $pk1$ and its storage $o1$, which are connected to the newly created table $t1$ via *secondary links*. Production $p^{(t)}$ is applicable in any situation, as it has no required context elements.

Productions "createFirstProperty" $p^{(fp)}$ and "createNextProperty" $p^{(np)}$ create the first property belonging to a type and the next properties belonging to a type respectively. Both TGG productions are different combinations of the patterns depicted in Fig. 3.4. They are only applicable if a class $c1$ and a table $t1$ exist that are already related via a TGG link $l2$. The

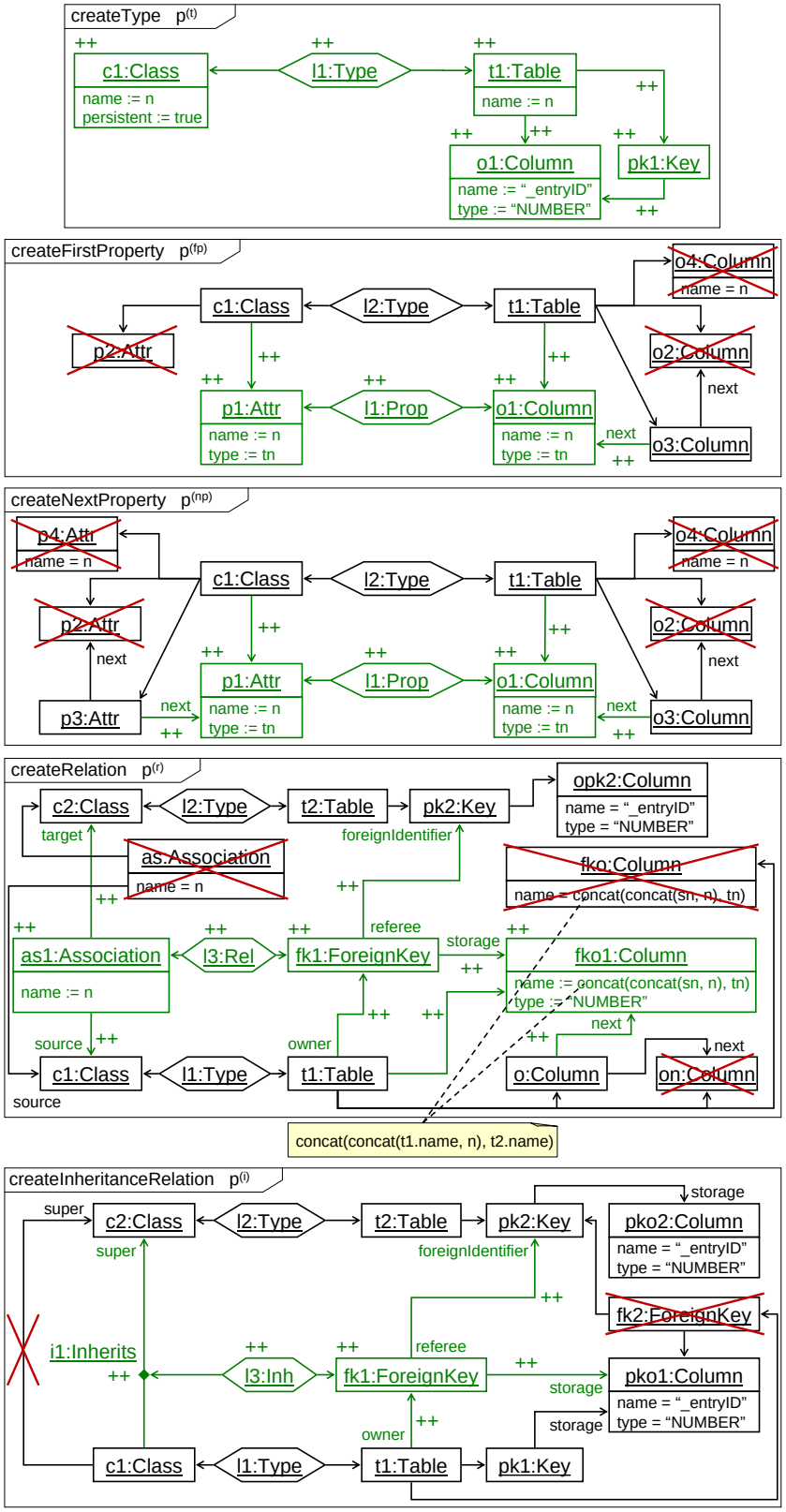


Figure 4.6.: TGG productions of TGG_{CDDS} that create types, properties, relationships, and inheritance structures.

correspondence and target component of both productions $p^{(fp)}$ and $p^{(np)}$ are identical. In conjunction, the source components of $p^{(fp)}$ and $p^{(np)}$ ensure that each class containing an attribute has exactly one attribute without predecessor and at most one attribute without successor. Production $p^{(fp)}$ is only applicable once for every related class and table because the NAC in the source component ensures that the production is only applied if the matched class does not already contain an attribute. Production $p^{(fp)}$ blocks its own further application in the context of $c1$ because it adds an attribute to $c1$ which results in a blocking NAC. But the newly created attribute $p1$ and column $o1$ both have no successor and, therefore, enable production $p^{(np)}$ to be applied from now on. Note that the source and target components of production $p^{(np)}$ are symmetric, i.e., have a one-to-one correspondence relationship. That is, the construction specification of "next" properties of source and target domain is identical up to isomorphism. Therefore, the types *Attr* and *Column* and the associations *Contains* and *Precedes* of both languages are mapped *one to one*.

Production "createRelation" $p^{(r)}$ creates relationships between two types. The primary nodes $as1$ and $fk1$, which identify a relationship in CD and DS, correspond to each other. The links which end at *source* and *owner* as well as the links that end at *target* and *foreignIdentifier* correspond to each other. The target component has an additional secondary object—column $fko1$—which is connected to the foreign key and added as last column to table $t1$. Therefore, the mapping of relationships in both languages is *not one to one* due to these additional secondary elements. Note that the target component contains a NAC which blocks the application if table $t1$ (the source of the relation) already contains a column with the same name, i.e., a relation with the same name has already been created between $t1$ and $t2$. Likewise, the source component restricts associations that relate $c1$ and $c2$ to have unique names. The name of column $fko1$ which encodes the name of the relation will become interesting when deriving backward translation rules from TGG_{CDDS} (cf. Sect. 5.2.3). This is due to the fact that the concat operations which are used to calculate the name of column $fko1$ depend on the names of source table $t1$ and target table $t2$—static information used by the production—and the given TGG parameter n—dynamic information.

Finally, production "createInheritanceRelation" $p^{(i)}$ creates inheritance structures between two types. The primary TGG link $l3$ relates the *primary link* $i1$ of the source component with the *primary object* $fk1$. The mapping of inheritance relations is *not one to one*—but *nearly one to one*—due to the additional *secondary link* which connects $fk1$ with $pko1$ in the target component.

4.5. Simultaneous Evolution of Graph Triples

We will now discuss the simultaneous evolution of graph triples by applying TGG productions of TGG_{CDDS} to an empty graph triple. The resulting graph triple GT_5 (cf. Fig. 4.7) is an element of the language of the just introduced TGG. Its source and target graph components have already been discussed as valid models of CDDS in Sect. 3.3.4 (cf. package 3.8(d) and database schema 3.8(e)). We have marked secondary objects with a dark background color to better distinguish them from primary objects. The graph triple is produced by applying production $p^{(t)}$ to the empty graph triple twice and afterwards production $p^{(r)}$ to the

resulting graph triple. Finally, production $p^{(fp)}$ is applied twice. Thus, GT_5 is produced by sequence $SEQ_5 = (p^{(t)}, p^{(t)}, p^{(r)}, p^{(fp)}, p^{(fp)})$. In the following we will abbreviate distinct TGG productions by their superscripts.

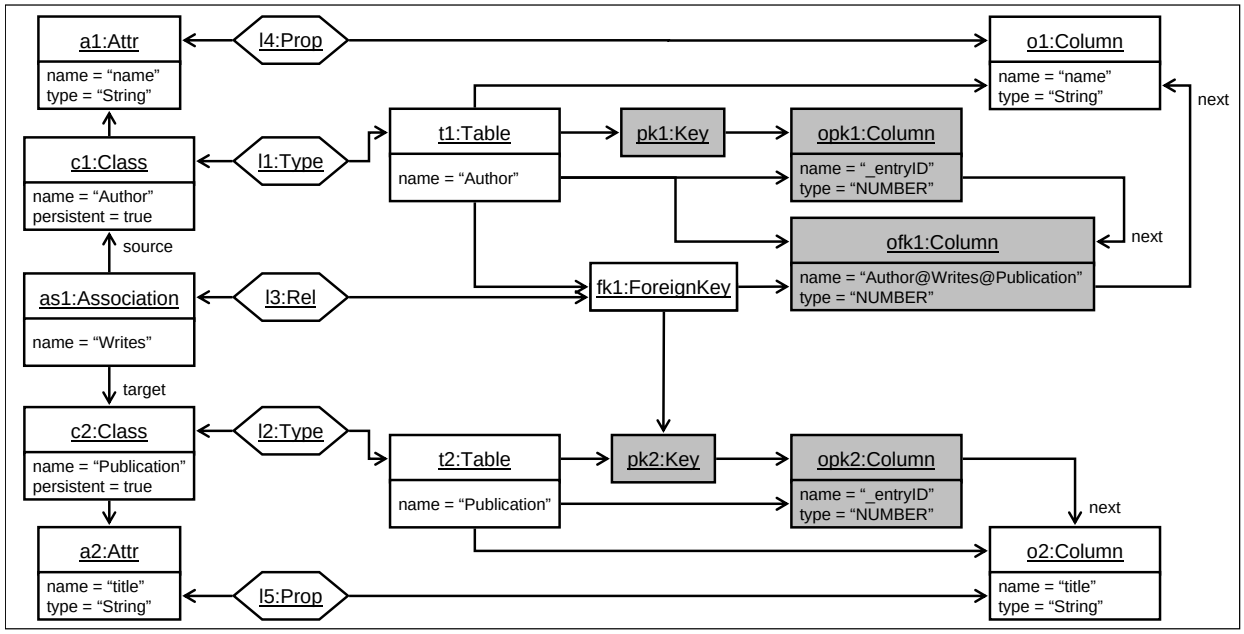


Figure 4.7.: Schema compliant graph triple GT_5 produced by TGG_{CDDS}.

First, production (t) simultaneously creates class $c1$ and table $t1$ (together with the secondary objects $pk1$ and $opk1$) and relates them via TGG link $l1$. Next, the second application of production (t) creates class $c2$ and table $t2$ (together with the secondary objects $pk2$ and $opk2$) and relates them via TGG link $l2$. Note that the context in which a production is applied is important: the context in which production (r) is applicable now is either $(c1 \leftarrow l1 \rightarrow t1,\ c2 \leftarrow l2 \rightarrow t2)$ or $(c2 \leftarrow l2 \rightarrow t2,\ c1 \leftarrow l1 \rightarrow t1)$. The source of the relationship is either $c1 \leftrightarrow t1$ or $c2 \leftrightarrow t2$. In this example, we choose $c1 \leftrightarrow t1$ as source of the relationship. The application of production (r) creates association $as1$ and foreign key $fk1$ and relates them via TGG link $l3$. Moreover, foreign key column $ofk1$, a secondary object, is created and linked to $fk1$, $t1$, and $opk1$.

Next, production (fp) is applicable in the context of $c1 \leftrightarrow t1$ or $c2 \leftrightarrow t2$ because neither classes $c1$ and $c2$ nor tables $t1$ and $t2$ contain elements at this moment. We choose $c1 \leftrightarrow t1$ and as a consequence attribute $a1$ and column $o1$ together with TGG link $l4$ are created. From now on, production (fp) is not applicable in the context of $c1 \leftrightarrow t1$ due to the NAC in the source component that blocks because class $c1$ already contains an attribute. Therefore, the second application of production (fp) only matches in the context of $c2 \leftrightarrow t2$ and adds a new attribute $a2$ and a new column $o2$ to the graph triple which are related via TGG link $l5$.

This leads to *production application sequence*
$SEQ_5 = (p^{(t)}@\emptyset,\ p^{(t)}@\emptyset,\ p^{(r)}@(c1 \leftrightarrow t1, c2 \leftrightarrow t2),\ p^{(fp)}@(c1 \leftrightarrow t1),\ p^{(fp)}@(c2 \leftrightarrow t2))$
that contains additional match information and the final situation depicted in Fig. 4.7.

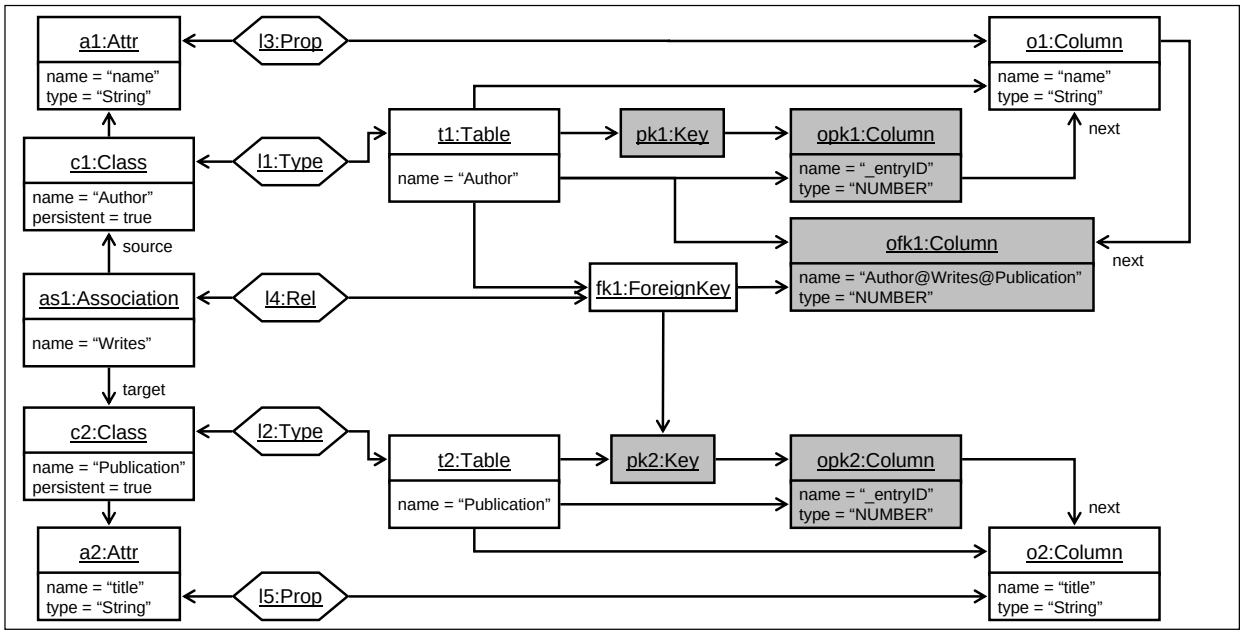


Figure 4.8.: Schema compliant graph triple GT_5^* produced by TGG_{CDDS}.

The order of columns in GT_5 is determined by the sequence of production applications SEQ_5. The ordering is $(opk1, ofk1, o1)$ in table $t1$ and $(opk2, o2)$ in table $t2$. If SEQ_5 is changed such that production (fp) is applied in the context of $c1 \leftrightarrow t1$ before production (r) is applied, this leads to sequence
$SEQ_5^* = (p^{(t)}@\emptyset,\ p^{(t)}@\emptyset,\ p^{(fp)}@(c1 \leftrightarrow t1),\ p^{(r)}@(c1 \leftrightarrow t1, c2 \leftrightarrow t2),\ p^{(fp)}@(c2 \leftrightarrow t2))$.

In this modified sequence of production applications, the order of the columns in tables $t1$ and $t2$ is $(opk1, o1, ofk1)$ and $(opk2, o2)$ respectively leading to graph triple GT_5^* (cf. Fig. 4.8). We consider graph triples GT_5 and GT_5^* to be semantically equivalent according to TGG_{CDDS} because the relative order of attributes in classes $c1$ and $c2$ is not destroyed and the relative order of columns of a relational database does not matter in a "pure" relational calculus.

4.6. Language Translators based on TGGs

This section gives a general overview to language translators based on triple graph grammars. A detailed discussion of the translation process, a control algorithm, and the translators derived from TGG_{CDDS} follows in Chap. 6. For a detailed description of the derivation process of translators we refer to Sect. 5.2.3.

A TGG can be compiled into a pair of *forward and backward graph translators* (*FGTs* and *BGTs*). The generated translators take a graph of the *input domain*, either source or target, and produce a graph triple that consists of the given input graph, the corresponding graph of the *output domain*, either target or source, and the correspondence graph which connects the related source and target graph elements.

A translator mainly consists of a set of *graph translation rules* and an algorithm that controls the stepwise translation of a given input graph into the related output graph. Each forward/backward *graph translation rule* (FGT/BGT rule), often called *operational rule*, is directly derived from a single TGG production. Therefore, TGG productions are split into sets of *local rules* and the aforementioned *translation rules* [Sch94] (cf. Sect. 5.2.3 for a formal view of the splitting process). Local rules generate graphs of the input domain ensuring that only valid graphs are produced. Hence, local rules are applicable only if NACs are not violated.

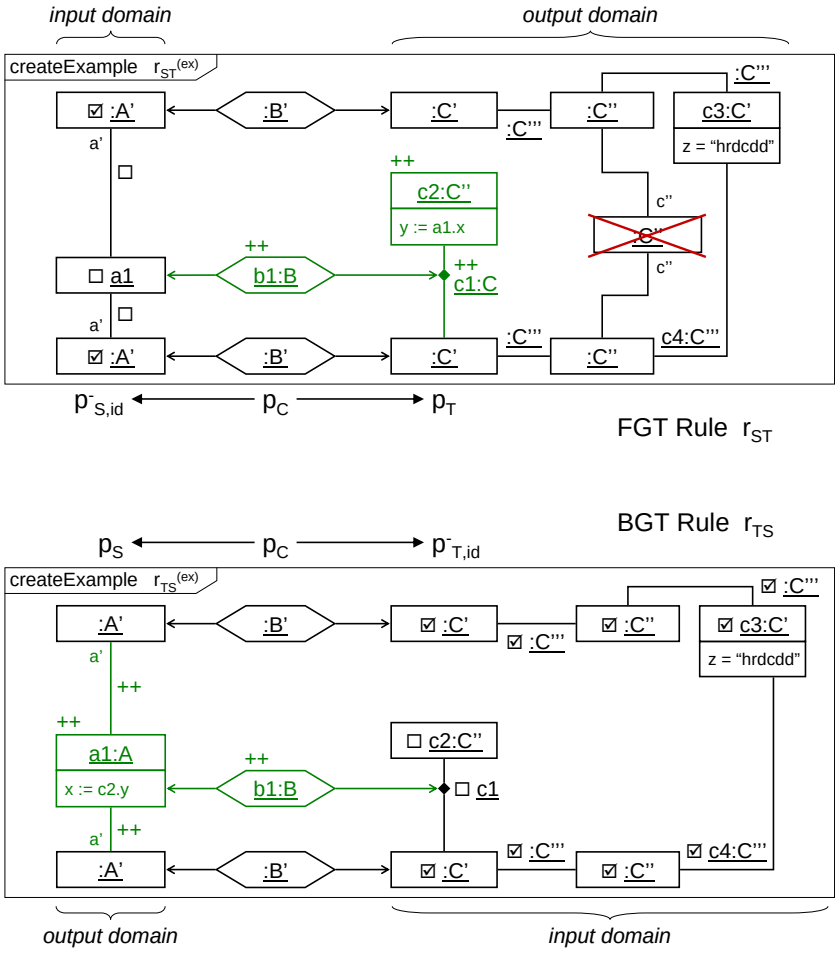


Figure 4.9.: Abstract example of forward and backward translation rules.

Speaking in terms of the modeling domain, a graph translator constructs an output model that corresponds to the input model. For example, if the class diagram contained in package 3.8 (d) is given as input to an FGT derived from TGG_{CDDS} then either GT_5 or GT_5^* (cf. Figs. 4.7 and 4.8), which both contain a corresponding database schema, is produced by the translator. Likewise, if database schema 3.8 (e) is given as input to a BGT derived from TGG_{CDDS} then GT_5, which contains a corresponding class diagram, is produced by the translator. Consequently, the sequence of translation rule applications is not necessarily

identical to the sequence of TGG production applications (cf. Sect. 4.5). But nevertheless, a translator produces an output which corresponds to the input model and is somehow equivalent—according to the specified TGG—to the output model produced by an according sequence of TGG production applications.

Figure 4.9 depicts the *FGT rule* $r_{ST}^{(ex)}$ and the *BGT rule* $r_{TS}^{(ex)}$ derived from TGG production $p^{(ex)}$ (cf. Fig. 4.5). Likewise to TGG productions, translation rules consist of source, correspondence, and target graph components. The elements of the input domain's graph component are read-only as they have been created earlier by a corresponding local rule. Consequently, translation rules only produce elements of the output and correspondence domain. Empty checkboxes in the input component denote elements of the input component that are created by the corresponding TGG production. In a translation rule, these elements are *to-be-translated elements* and must not be translated yet, i.e., no corresponding elements in the output domain have been created by another translation operation beforehand. A translator will mark all elements of the input graph that have been matched by the to-be-translated elements of the input component as "translated"—i.e., add a check mark to the checkbox that indicates the translation status—right after a translation rule has been applied successfully.

Elements that were context elements in a TGG production must be translated before a rule is applicable. This is denoted as enabled checkbox placed next to or inside these elements. NACs of the input domain may be omitted under certain conditions, as we will learn in Sect. 5.2.3, whereas NACs of the output domain are retained.

A translation algorithm applies the operational rules to the input graph such that it simulates the simultaneous evolution of the computed graph triple with respect to the given set of TGG productions. Therefore, a translator must be able to determine the order in which elements of the input graph would have been created by a sequence of TGG productions. Interleaved with the stepwise computation of a sequence of TGG productions and the resulting derivation of the input graph, the corresponding sequence of operational rules is executed by a translator to generate the related output and correspondence graph instances.

Guessing the proper choice of translation rule applications is one of the difficulties that arise when the simultaneous evolution of graph triples is simulated by a translator. This is due to the fact that in the general case multiple decision points exist while determining the sequence of translation rules [Sch94]. If a translator chooses a wrong path in the possible tree of sequences this will lead into a *dead-end* (i.e., wrong translation alternative which requires *backtracking*). Consequently, the translator has to track back to the wrong decision point and try the next alternative. In general, the computation of an appropriate sequence of rules requires a graph grammar parsing algorithm with exponential runtime behavior [RS97], i.e., such parsing algorithms are not efficient in general.

4.7. Fundamental Properties of TGGs and Translators

In Sect. 3.5 we discussed properties of bx-languages and translators for the general case. Now, we will discuss *expressiveness* in the context of triple graph grammars and *efficiency*, *correctness*, and *completeness* in the context of derived forward and backward translators.

Expressiveness requires that the TGG formalism is able to capture all important "mapping techniques" between studied pairs of languages. Common mapping scenarios include mapping of objects and relationships between objects, i.e., links. Relationships can be one-to-one (cf. relationships of LM_{CD} and LM_{DS} depicted in Fig. 3.2, e.g., *Precedes* and *StoresKeysIn*), one-to-many (e.g., relationships *Contains* and *RelatesToForeignEntriesVia*), many-to-one (e.g., relationships *EndsAt* and *StartsAt*), or many-to-many (e.g., relationship *Inherits*). Such relationships can be used in the specification of TGG productions as shown in Sect. 4.4 to relate corresponding concepts of two languages. (Negative) application conditions are also required in practice when specifying mapping definitions with triple graph grammars, e.g., to avoid creating graph triples that violate their schema definition. Moreover, secondary elements are required, e.g., to encode additional information or to realize features not natively supported by a language. Finally, basic operations—like operation "concat"—are required when calculating attribute values of objects.

These features can be added more or less easily to the triple graph grammar formalism (cf. Chap. 5). But in addition to adding these features to the TGG formalism, they also have to be added to the level of translators without destroying the fundamental properties *efficiency*, *correctness*, and *completeness* (cf. Chap. 6).

Especially, *efficiency* of translators competes against the properties *correctness* and *completeness*. After more than 15 years of TGG research activities the TGG community still faces, e.g., problems to handle TGGs with NACs appropriately, i.e., to find the right compromise between expressiveness of TGG productions on the one hand and the correctness, completeness, and efficiency properties of derived translators on the other hand [SK08, KLKS10]. According to [KLKS10], efficient translators have polynomial space and time complexity $O\left(m \times n^k\right)$ with m = number of rules, n = size of input graph, and k = maximum number of elements of a rule. This requirement is based on two worst case assumptions:

(1) n^k is the worst-case complexity of the pattern matching step of a graph translation rule with k elements.

(2) Without starting the pattern matching process for a selected rule we cannot determine whether this rule can be used to translate a just regarded element.

As a consequence we require that derived translators do somehow process the elements of an input graph in a given order such that no element has to be regarded and translated more than a constant number of times. Selecting always somehow the "right" translation rules we do not have to explore multiple translation alternatives using, e.g., a depth-first backtracking algorithm for that purpose.

Correctness is guaranteed if a translator translates an input graph into a graph triple GT that is always an element of the language $\mathcal{L}(TGG)$ defined by the TGG. *Completeness* demands that for every graph triple GT that is an element of $\mathcal{L}(TGG)$, a translator is able to produce this graph triple (or an equivalent one) given the graph of the graph triple, which belongs to the translator's input domain.

As a consequence the original TGG approach [Sch94] is continuously extended such that it supports, e.g., handling of attributed typed graphs [EEE+07] as well as the definition of productions with NACs [SK08, EHS09, EEHP09, KLKS10].

In the next chapter we will prepare the building blocks that will allow us to build *efficient*, *correct*, and *complete* translators based on triple graph grammars that support the features related to expressiveness discussed above. Chapter 6 then explains *how* such translators based on triple graph grammars are built.

5. Extended Triple Graph Grammars

[Sch94] formally introduces simple triple graph grammars and proves that forward and backward translators can be derived from TGGs. In addition, it sketches an abstract algorithm that is able to realize translators. Moreover, it sketches extended triple graph grammars which improve the expressiveness of TGGs. According to [Sch94] such extended TGGs are needed if the triple graph grammar formalism should be of any practical relevance. But such extensions should not violate the fundamental properties of derived translators, i.e., termination, correctness, completeness, and the overall goal of efficiency. [Sch94] discusses the three steps to extended TGGs:

(1) Introduce vertex/node and edge labels for each regarded class of graphs, i.e., introduce type graphs as discussed in Sect. 4.1.

(2) Deal with attributes for vertices/nodes (and edges) and *attribute value parameters*, i.e., TGG parameters as discussed in Sect. 4.3.

(3) Introduce additional means for restricting the applicability of productions, i.e., a sufficiently large set of application conditions.

In Chap. 5 we will tackle these steps such that we are able to derive translators in Chap. 6 that still fulfill the fundamental properties of TGG-based translators. We start discussing steps (1) and (2) in Sect. 5.1. Afterwards, we discuss step (3) in Sect. 5.2 where we introduce a certain subset of negative application conditions to the TGG formalism. We will continue with further discussions that are related to the derivation process of translation rules and the problems that arise when constructing a translation algorithm that controls these translation rules at *translator level* in order to simulate simultaneous evolution processes at *TGG level*. Therefore, Sect. 5.3 introduces the so-called *dangling edge condition*.

5.1. Labels and Attributes

According to [EEPT06], labeled graphs can be considered as special case of typed graphs. Therefore, the theory of graphs and typed graphs can also be applied to labeled graphs (cf. Sect. 2.6). Hence, we will base the extended triple graph grammar approach on typed graphs (cf. Def. 12 in Sect. 4.1) to achieve step (1). Consequently, we demand in our graph triple rewriting formalism (cf. Def. 17 in Sect. 5.2.2) that all morphisms between graph components of GT and GT' are *type preserving* when rewriting a typed (and labeled) graph triple GT into another graph triple GT' in a direct derivation. Furthermore, we know from [EEPT06] that we are able to "attach" labels to typed graph triples using a distinguished type graph which

simulates labels. That is, if required we will simulate labels utilizing an additional *label graph* for each graph component of a graph triple.

Concerning step (2), we will simulate attributes, slots, and values inside a graph by utilizing a mapping definition, e.g., utilizing the definition that we used to map the concept of attributes/properties from the world of models to the world of graphs (cf. Fig. 2.23). That is, we use a reserved graph structure—i.e., unambiguous graph structure—for representing attributes, slots, and values in a graph[1] for representing the concept of attributes inside a graph. A TGG production that creates slots and values or compares certain attribute values will then internally operate on that reserved graph structure. Therefore, we require that each attribute belonging to a class is uniquely determinable, e.g., by name. In addition, we will always treat such simulated attributes, slots, and values as secondary elements inside a TGG production. Finally, TGG parameters used inside a TGG production are specifically handled when deriving operational rules from a TGG production (cf. Sects. 5.2.3 and 6.9).

5.2. Formalization of Constrained TGGs with NACs

In the preceding chapter we have informally introduced TGGs with NACs (cf. Sect. 4.3). In order to realize step (3) of extended TGGs, we will now formally introduce a certain subset of negative application conditions. In general, application conditions allow to restrict the application of productions and are important for increasing the expressive power of graph transformation systems [EEPT06, p. 64]. A NAC will block the application of a production if a match m of the left-hand side violates the conditions specified by the NAC. We have already learned that NACs are additional preconditions that must be satisfied so a production is applicable. NACs are, e.g., used to prohibit the construction of graph triples that violate constraints defined in the schema of the source and target domain. Likewise, (positive) application conditions (*PACs*) prevent application of a production if a match m of the left-hand side does not satisfy the conditions specified by the PAC. Unfortunately, the concept of negative application conditions as well as the concept of (positive) application conditions is not part of the original TGG formalism, but is needed in practice to increase the expressiveness of the TGG language. Simple PACs can be realized by extending the context of a production. For example, a transformation step will check for the existence of certain nodes and edges, if these elements are specified as context elements of the production, before the production is applied. In the following, we will increase the expressiveness of the TGG formalism by supporting a certain category of negative application conditions.

The introduction of NACs into the world of TGGs also introduces a new kind of rule application conflict that has to be taken into account by graph translators. That is, enabling NACs in a TGG production results in potential rule application conflicts. Essentially, the definition and application of TGG productions is restricted in such a way that a rule application control algorithm never has to resolve rule application conflicts by making an arbitrary choice. Potential rule application conflicts of graph translators are as follows:

[1] This allows to use high-level operations "getAttributes(Class)" and "getSlots(Object)" that are able to determine attributes, slots, and values inside a given graph.

1. A *positive/negative rule application conflict* of two operational rules $r1$ and $r2$ w.r.t. specific redexes exists if $r1$ creates or translates a graph element that is forbidden by a NAC of $r2$.

2. A *positive/positive rule application conflict* of two operational rules $r1$ and $r2$ w.r.t. specific redexes exists if both rules compete to translate the same graph element.

For reasons of efficiency rule application conflicts should be avoided if possible. In order that an algorithm may not run into positive/negative conflicts we replace NACs on the input graph side of a translation by graph constraints. Therefore, we will identify a sort of TGG productions with NACs which do not lead to positive/negative FGT/BGT rule application conflicts for any input graph. These TGG productions are called integrity-preserving productions (cf. Sect. 5.2.1). This is a first step towards our goal to eliminate all kinds of FGT/BGT rule application conflicts and thereby to guarantee completeness of derived translation functions. For this purpose we extend the TGG formalism as introduced in [Sch94] by NACs that are used to preserve the integrity of graph triples (i.e., resulting graph triples never violate constraints—neither temporary) without destroying the fundamental propositions proved in [Sch94]. Therefore, TGGs will operate on typed constrained graphs and support NACs in a way that derived translators do not violate the mentioned compatibility properties correctness and completeness. Permitted NACs will be ignored on the input graph of translation rules assuming that integrity violations of input graphs are captured before a translation process starts.

Positive/positive rule application conflicts are then eliminated by inspecting the context of those nodes more closely that are just translated by a given rule (cf. Sect. 5.3). Inspired by the definition of the double-pushout (DPO) graph grammar approach [EEPT06] (cf. Sect. 2.7) a new kind of "dangling edge condition" is, therefore, introduced in Sect. 5.3 that blocks the translation of nodes with afterwards still untranslated incident edges under certain conditions. Consequently, an algorithm does not have to make decisions which lead into dead-ends because only one rule—more precisely one kind of rules—remains applicable.

5.2.1. Integrity Preserving Productions

Based on the definitions in Sect. 2.7 we will now define *graph productions with NACs* and graph rewriting with NACs, which are then used for the definition of TGGs that generate triples of typed and constrained graphs in Sect. 5.2.2. We will start with the definition of monotonic productions with NACs which is based on Def. 7 (cf. Sect. 2.7).

Definition 13. *Monotonic Graph Productions with NACs.*
The set of all monotonic productions $\mathcal{P}(TG, C)$ with negative application conditions $\mathcal{N}$ for a type graph TG with a set of constraints $\mathcal{C}$ is defined as follows:
$(L, R, \mathcal{N}) \in \mathcal{P}(TG, \mathcal{C})$ *iff*
(1) $L, R \in \mathcal{L}(TG, \mathcal{C}) \wedge L \subseteq R$
(2) $\mathcal{N} \subseteq \mathcal{L}(TG) \wedge \forall N \in \mathcal{N}{:}N \supseteq L$

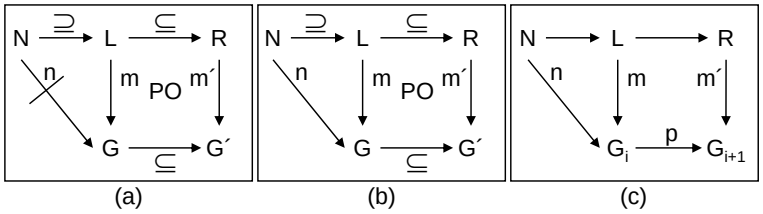


Figure 5.1.: Diagrams used in Def. 14, Def. 15, and in proof of Corollary 3.

Now we define graph rewriting for the just introduced monotonic productions. Such productions rewrite a graph G into a graph G' almost like regular productions but only if no match is found for any of the production's negative application conditions in the host graph G.

Definition 14. *Graph Rewriting for Monotonic Productions with NACs.*
A monotonic production $p := (L, R, \mathcal{N}) \in \mathcal{P}(TG, \mathcal{C})$ *rewrites a graph* $G \in \mathcal{L}(TG)$ *into a graph* $G' \in \mathcal{L}(TG)$ *with a redex (match)* $m : L \to G$*, i.e.,* $G \overset{p@m}{\rightsquigarrow} G'$ *iff*
(1) $m' : R \to G'$ *is defined by building the pushout diagram presented in Fig. 5.1 (a)*
(2) $\neg(\exists N \in \mathcal{N}, n : N \to G\text{: } n|_L = m)$*, i.e., there exists no* N *such that mapping* n *is identical to* m *w.r.t. the left-hand side graph* L
(3) all morphisms are type preserving

We will limit productions with NACs to so-called *integrity-preserving productions* in Def. 15 such that NACs are only used to prevent the creation of graphs which violate the set of constraints $\mathcal{C}$. These productions have the important properties that (1) given a valid input graph, a valid output graph is produced, (2) if productions where NACs are eliminated produce a valid graph then the input graph is also valid, and (3) a production that would block due to a NAC otherwise would always produce an invalid graph. Due to contraposition of (1) all invalid output graphs are derived from invalid input graphs. So, integrity-preserving productions produce only invalid output graphs if the input graph was already invalid. Moreover, contraposition of (2) (i.e, (2*)) states that invalid input graphs result in invalid output graphs even if NACs are eliminated from a production; i.e., productions with or without NACs do not repair invalid graphs.

Definition 15. *Integrity-Preserving Productions.*
Let p *be a monotonic production* $(L, R, \mathcal{N}) \in \mathcal{P}(TG, \mathcal{C})$ *and* $p^- := (L, R, \emptyset)$ *being the corresponding production of* p *where all negative application conditions have been eliminated. Then,* p *is integrity-preserving iff*
(1) $\forall G, G' \in \mathcal{L}(TG) \wedge G \overset{p}{\rightsquigarrow} G'\text{: } G \in \mathcal{L}(TG, \mathcal{C}) \Rightarrow G' \in \mathcal{L}(TG, \mathcal{C})$
(2) $\forall G, G' \in \mathcal{L}(TG) \wedge G \overset{p^-}{\rightsquigarrow} G'\text{: } G' \in \mathcal{L}(TG, \mathcal{C}) \Rightarrow G \in \mathcal{L}(TG, \mathcal{C})$
(3) $\forall N \in \mathcal{N}$*: the existence of the diagram depicted in Fig. 5.1 (b) with type preserving morphisms* $n|_L = m = m'|_L$ *implies* $G' \in \overline{\mathcal{L}}(TG, \mathcal{C})$

Now, we show for TGG_{CDDS} that the source and target production components (i.e., class diagram and database schema components; cf. Fig. 4.6) satisfy the conditions of Def. 15.

In order that productions of TGG_{CDDS} are integrity-preserving, the following OCL invariant has to be added to the TGG schema.

$inv_{DS:F:inheritance}$ Only one foreign key element may be added to a database schema to express the inheritance relationship of two tables. Such a foreign key uses the primary key column of the subtable as its storage, i.e., the name of the storage is "_entryID". Every two foreign keys that have the same owner, i.e., subtable, must have different foreign identifiers, i.e., supertables.
context ForeignKey inv: ForeignKey.allInstances()->select(storage.name = "_entryID")->forAll(fk1, fk2 | fk1 <> fk2 and fk1.owner = fk2.owner implies fk1.foreignIdentifier <> fk2.foreignIdentifier)

Now, all productions satisfy condition (1), i.e., given a valid graph, productions where NACs are not removed produce only graphs that are valid according to the TGG schema (cf. Sect. 3.3.1 and Sect. 4.2) and the set of constraints of TGG_{CDDS} (cf. Sect.3.3.2):

- Source and target components of TGG production "createType" satisfy condition (1). The class diagram component produces one single class which is a valid situation according to LM_{CD}.

  The database schemata component produces a table, a primary key and a column. The latter two objects are produced and linked with the new table, which is necessary in order to satisfy the lower bound "1" of the multiplicity constraints of associations *IdentifiesEntriesWith* and *StoresKeysIn* of LM_{DS} (cf. Fig. 3.2). Moreover, the new column is the "first" and the "last" column contained by the table, i.e., every table initially contains one column.

- Source component of TGG production "createFirstProperty" creates an attribute $p1$ and adds it as the first attribute to a given class. OCL invariants $inv_{CD:C:attr:first}$ and $inv_{CD:C:attr:last}$ demand that at most one "first" and one "last" attribute are contained in a class. These constraints are not violated by adding this "first" attribute $p1$ which is also the one and only "last" attribute after application of the production.

  The database component of TGG production "createFirstProperty" selects the last column $o3$ of the given table and adds a new column $o1$. In order to not violate OCL invariant $inv_{DS:T:col:last}$ the new column $o1$ is set as successor of the former last column $o3$. This is necessary because otherwise the table would contain two last columns $o1$ and $o3$ which is forbidden by $inv_{DS:T:col:last}$. By setting $o1$ as successor of $o3$, column $o3$ is no longer a last column, which keeps the number of "last" columns at a constant level, i.e., ensures that every table contains at most one "last" column, Moreover, the name of $o1$ is guaranteed to be unique in the given table, which is demanded by $inv_{DS:T:col:unique}$ and ensured by the NAC containing column $o4$.

- Source and target components of TGG production "createNextProperty" are similar to the database component of TGG production "createFirstProperty" and, therefore, satisfy condition (1) for the same reasons.

- Source component of TGG production "createRelation" creates an association which is linked to a source and a target class. This ensures multiplicity constraint "1" of the association's source and target class.

Similarly, the target component creates a foreign key, a storage column, and links these elements. This ensures multiplicity constraints "1". Moreover, the new column is guaranteed to be set as new "last" column.

- Source component of TGG production "createInheritanceRelation" creates a new inheritance link between two classes, which always produces a valid graph according to LM_{CD}. The target component produces a new foreign key which is linked appropriately according to the multiplicity constraints of LM_{DS}.

If productions "createFirstProperty", "createNextProperty", "createRelation", and "createInheritanceRelation" without NACs produce valid output graphs under certain conditions then the input graph was also valid due to the fact that none of the productions is able to repair invalid graphs even if their NACs are ignored. Therefore, TGG_{CDDS} satisfies condition (2) (and its contraposition (2*)) of Def. 15 because:

- Source component of TGG production "createFirstProperty" would produce even more (invalid) attributes without predecessor and successor if NACs are ignored.
- Target component of TGG production "createFirstProperty" and source and target component of TGG production "createNextProperty" do not reduce the number of attributes and columns without predecessor. Moreover, they preserve or even increase the number of attributes and columns without successor that violate $inv_{CD:P:n:mult}$ and $inv_{DS:P:n:mult}$.
- Likewise, target component of TGG production "createRelation" does not reduce the number of columns without successor.
- Source component of TGG production "createRelation" would produce even more associations with a non-unique name if NACs are ignored.
- Components of TGG production "createInheritanceRelation" increase the number of inheritance links and foreign keys that indicate an inheritance relationship respectively. If a resulting database schema is valid, i.e., constraint $inv_{DS:F:inheritance}$ is fulfilled, then the original schema was also valid because no inheritance structure existed between the given tables.

Condition (3) of Def. 15 is satisfied because:

- A blocking NAC of source component of production "createFirstProperty" prevents the production of an additional attribute without successor.
- Blocking NACs in target component of production "createFirstProperty" and source and target component of production "createNextProperty" prevent violation of the multiplicity constraint of a "Precedes" link $inv_{CD:P:n:mult}$ and $inv_{DS:P:n:mult}$. Therefore, violation of $inv_{CD:C:attr:last}$ and $inv_{DS:T:col:last}$ is prevented. Moreover, the second NACs prevent violation of invariants $inv_{CD:C:attr:unique}$ and $inv_{DS:T:col:unique}$, i.e., attributes and columns must have unique names.

- Similarly, a blocking NAC in source and target component of production "createRelation" prevents violation of invariants $inv_{CD:Pkg:assoc:unique}$, $inv_{DS:T:col:unique}$, and $inv_{DS:T:col:last}$.
- A blocking NAC of source component of production "createInheritanceRelation" prevents creation of multiple links of the same type between two objects as this is not supported by the utilized framework. Likewise, the NAC in the target component blocks if the resulting graph would violate constraint $inv_{DS:F:inheritance}$.

Therefore, the set of productions of TGG_{CDDS} is *integrity-preserving*. The productions of TGG_{CDDS} that make use of NACs guarantee that no invalid graph triples are produced by the productions. An invalid graph triple would either violate a multiplicity constraint or an OCL invariant of the TGG schema (cf. Figs. 4.3 and 3.2 and constraints of LM_{CD} and LM_{DS} discussed in Sect. 3.3.2).

Productions that destroy the integrity of graphs—i.e., *integrity-destroying productions*—as well as *integrity-restoring productions* are not supported in our approach.

Finally, Def. 16 states how graph grammars produce constrained typed graphs.

Definition 16. *Language of Typed and Constrained Graph Grammars.*
A graph grammar $GG := (TG, \mathcal{C}, \mathcal{P})$ *over a type graph* TG*, a set of constraints* $\mathcal{C}$*, and a finite set of integrity-preserving productions* $\mathcal{P} \subseteq \mathcal{P}(TG, \mathcal{C})$*, with* $G_\emptyset$ *being the empty graph, generates the following language of graphs*
$\mathcal{L}(GG) := \{G \in \mathcal{L}(TG, \mathcal{C}) | G_\emptyset \overset{p_1}{\rightsquigarrow} G_1 \overset{p_2}{\rightsquigarrow} ... \overset{p_n}{\rightsquigarrow} G_n = G \text{ with } p_1, ..., p_n \in \mathcal{P}\}$

With the following corollaries we show that we can safely remove NACs from an integrity-preserving production and check after production application if constraints are satisfied.

Corollary 1 states that the language that is generated by a graph grammar GG as defined in Def. 16 (i.e., the graphs that are producible by the grammar) is a subset of the set of graphs of type TG that fulfill the given set of constraints $\mathcal{C}$.

Corollary 1. $\mathcal{L}(GG) \subseteq \mathcal{L}(TG, \mathcal{C})$

Proof. Follows from Def. 15 and directly from Def. 16. □

Furthermore, Corollary 2 states that the language $\mathcal{L}(GG^-)$ generated by a graph grammar where NACs of productions have been eliminated contains at least the same graphs as the language $\mathcal{L}(GG)$ generated by this graph grammar with NACs.

Corollary 2. *Let* GG^- *be a graph grammar derived from a graph grammar* GG*, where all negative application conditions of productions have been eliminated. Then* $\mathcal{L}(GG^-) \supseteq \mathcal{L}(GG)$.

Proof. Due to the fact that a valid application of a production p with NACs is also a valid application of the production p^- where NACs are ignored, $\mathcal{L}(GG^-)$ is at least as large as $\mathcal{L}(GG)$. □

Moreover, Corollary 3 states that $\mathcal{L}(GG)$ is the intersection of the graphs producible by $\mathcal{L}(GG^-)$ and the set of graphs of type TG that fulfill the given set of constraints $\mathcal{C}$.

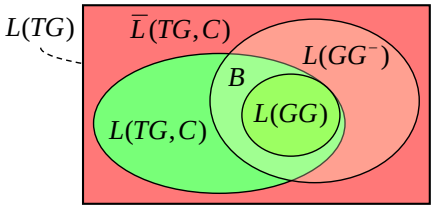


Figure 5.2.: Languages of graphs and languages of graphs generated by graph grammars.

Corollary 3. *Let GG^- be a graph grammar derived from a graph grammar GG as defined in Def. 16, where all negative application conditions of productions have been eliminated. Then* $\mathcal{L}(GG) = \mathcal{L}(GG^-) \cap \mathcal{L}(TG, \mathcal{C})$.

Proof. Due to Corollary 1 and Corollary 2 the intersection of sets of graphs defined by $\mathcal{L}(TG, \mathcal{C})$, $\overline{\mathcal{L}}(TG, \mathcal{C})$, $\mathcal{L}(GG)$, and $\mathcal{L}(GG^-)$ looks like depicted in Fig. 5.2.
Therefore, we only have to show that $\mathcal{B} := \mathcal{L}(GG^-) \cap \mathcal{L}(TG, \mathcal{C}) \backslash \mathcal{L}(GG)$ is empty. Let $G \in \mathcal{B}$, i.e., G is generated by a sequence of production applications

$$G_\emptyset \rightsquigarrow \ldots \rightsquigarrow G_i \overset{p^-}{\rightsquigarrow} G_{i+1} \rightsquigarrow \ldots \rightsquigarrow G$$

with $p = (L, R, \mathcal{N})$ being an integrity-preserving production of GG and $p^- = (L, R, \emptyset)$ being the corresponding production of GG^- such that $\exists N \in \mathcal{N}$ so the diagram depicted in Fig. 5.1 (c) commutes, i.e., p is blocked by N, but p^- rewrites G_i into G_{i+1}.
$\Rightarrow G_{i+1} \in \overline{\mathcal{L}}(TG, \mathcal{C})$. This is a direct consequence of Def. 15 (3), which requires that the application of p^- produces a graph G_{i+1}, which violates at least one constraint if the application of p is blocked by its NAC N.
$\Rightarrow G \in \overline{\mathcal{L}}(TG, \mathcal{C})$. This is a direct consequence of Def. 15 (2^*) because all graphs on the derivation path from G_{i+1} to G (including G) are invalid due to the fact that productions of GG^- preserve the property of a graph to violate some constraint. This leads to contradiction. □

As a consequence of Def. 15 and due to Corollary 3, we can either check NACs during the execution of a (TGG) production to prohibit the violation of graph constraints immediately or check potentially violated graph constraints after a sequence of graph rewriting steps that simply ignore NACs; for a more detailed discussion of the relationship of (positive) pre- and postconditions of graph transformation rules and graph constraints we refer to [HC07].

5.2.2. Constrained and Typed Triple Graph Grammars with NACs

Having introduced definitions and properties of graph grammars with NACs for languages of constrained typed graphs we now present the corresponding definitions of TGGs with NACs for constrained typed graph triples. We lift graph rewriting (cf. Def. 14) based on monotonic productions (cf. Def. 13) and integrity-preserving productions (cf. Def. 15) to graph triple rewriting in Def. 17.

Definition 17. *Integrity-Preserving Graph Triple Rewriting.*
Let $p := (p_S \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$ *be a* production triple with NACs *and*
(1) $p_S := (L_S, R_S, \mathcal{N}_S) \in \mathcal{P}(TG_S, \mathcal{C}_S)$ *be an* integrity-preserving production

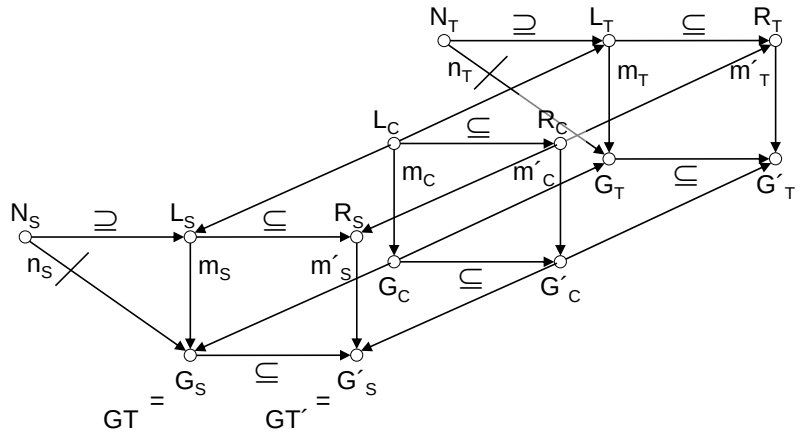


Figure 5.3.: Extended "Pair of Cubes" diagram utilized by Integrity-Preserving Graph Triple Rewriting.

(2) $p_C := (L_C, R_C, \emptyset) \in \mathcal{P}(TG_C, \emptyset)$ *be a* simple production
(3) $p_T := (L_T, R_T, \mathcal{N}_T) \in \mathcal{P}(TG_T, \mathcal{C}_T)$ *be an* integrity-preserving production
(4) $h_S : R_C \to R_S$, $h_S|_{L_C} : L_C \to L_S$ *and* *(5)* $h_T : R_C \to R_T$, $h_T|_{L_C} : L_C \to L_T$
The application of such a production triple to a graph triple GT produces another graph triple GT′, i.e., $GT \overset{p}{\rightsquigarrow} GT'$, *which is uniquely defined (up to isomorphism) by the existence of the extended "*pair of cubes*" diagram depicted in Fig. 5.3.*
This diagram consists of commuting square-like subdiagrams only and contains a pushout subdiagram for each application of a production component *(i.e,* p_S, p_C, *and* p_T*) to its corresponding* graph component.

The "pair of cubes" diagram depicts the application of a TGG production at a given typed host graph triple GT. The rewriting step is type preserving, i.e., the graph triple morphism between GT and the resulting graph triple GT' is part of a commuting "toblerone" diagram (cf. Fig. 4.2).

For the details of the definition and the proof that production triples applied to graph triples at a given redex always produce another graph triple uniquely defined up to isomorphism, cf. [Sch94]. NACs introduced here do not destroy the constructions and proofs introduced in [Sch94] due to the fact that they do not (further) influence the application of a production to a given graph (triple) after all NAC applicability checks have been executed. Based on the presented definitions we introduce *typed triple graph grammars* and their languages. For reasons of readability we omit the prefix "typed" throughout the rest of this contribution.

Definition 18. *Triple Graph Grammar and Triple Graph Grammar Language.*
A triple graph grammar *TGG over a triple of type graphs* (TG_S, TG_C, TG_T) *is a tuple* $(P, GT_\emptyset)$, *where* P *is the set of its TGG productions and* $GT_\emptyset$ *is the empty graph triple. The language* $\mathcal{L}(TGG)$ *is the set of all graph triples that can be derived from* $GT_\emptyset := (G_\emptyset \overset{\varepsilon}{\leftarrow} G_\emptyset \overset{\varepsilon}{\rightarrow} G_\emptyset)$ *using a finite number of TGG production rewriting steps.*

Similarly to Corollary 3, we can show that a triple graph grammar TGG^-, where all NACs (that prevent the creation of graph triples that violate graph constraints) are removed from

TGG productions, produces the same set of constrained graph triples that is produced by the unmodified triple graph grammar TGG.

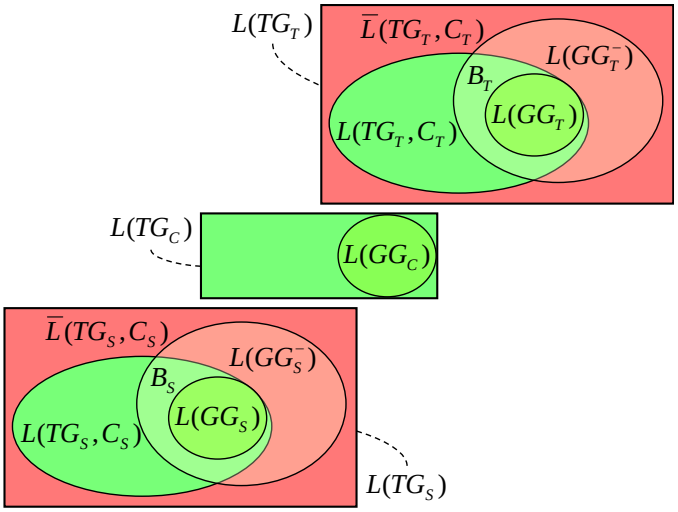


Figure 5.4.: Languages of graph triples and languages of graph triples that are generated by extended triple graph grammars.

Theorem 1. *With $\mathcal{L}(TGG)$ being the language of graph triples generated by a triple graph grammar TGG over (TG_S, TG_C, TG_T) we can show:*
(1) for all $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$:
$G_S \in \mathcal{L}(TG_S, \mathcal{C}_S)$, $G_C \in \mathcal{L}(TG_C)$, $G_T \in \mathcal{L}(TG_T, \mathcal{C}_T)$
(2) with TGG^- being the triple graph grammar derived from TGG
where all NACs of productions have been removed:
$(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$
$\Leftrightarrow (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG^-) \wedge (G_S, G_C, G_T) \in \mathcal{L}(TG_S, \mathcal{C}_S) \times \mathcal{L}(TG_C) \times \mathcal{L}(TG_T, \mathcal{C}_T)$

Proof. Follows from Def. 17 (which lifts graph to graph triple rewriting) and Corollaries 1 and 2. The proof that uses Fig. 5.4 is analogous to the proof of Corollary 3. □

5.2.3. Splitting of Production Triples with NACs

It is a direct consequence of Theorem 1 that checking of NACs can be replaced by checking integrity of generated graphs with respect to their sets of constraints and vice versa. This observation directly affects translators derived from a given TGG as follows: According to [Sch94], a production triple p may be split into pairs of production triples (r_I, r_{IO}), where r_I is an *(input-) local rule* and r_{IO} its corresponding *(input-to-output domain) translation rule*, with $GT \overset{p}{\rightsquigarrow} GT' \Leftrightarrow GT \overset{r_I}{\rightsquigarrow} GT_I \overset{r_{IO}}{\rightsquigarrow} GT'$. Forward translation is based on (r_S, r_{ST}), whereas (r_T, r_{TS}) is used in the reverse direction.

To rewrite the source graph only, the *source-local production triple*, i.e., *source-local rule* $r_S := (p_S \overset{\varepsilon}{\leftarrow} (\emptyset, \emptyset, \emptyset) \overset{\varepsilon}{\rightarrow} (\emptyset, \emptyset, \emptyset))$ is applied (cf. left-hand side of Fig. 5.5). To rewrite the

target graph only, the *target-local rule* $r_T := ((\emptyset,\emptyset,\emptyset) \overset{\varepsilon}{\leftarrow} (\emptyset,\emptyset,\emptyset) \overset{\varepsilon}{\rightarrow} p_T)$ is applied. The source-local and target-local rules of TGG_{CDDS} are equal to the productions depicted in Figs. 3.3, 3.4, 3.5, and 3.6 (cf. Sect. 3.3.3).

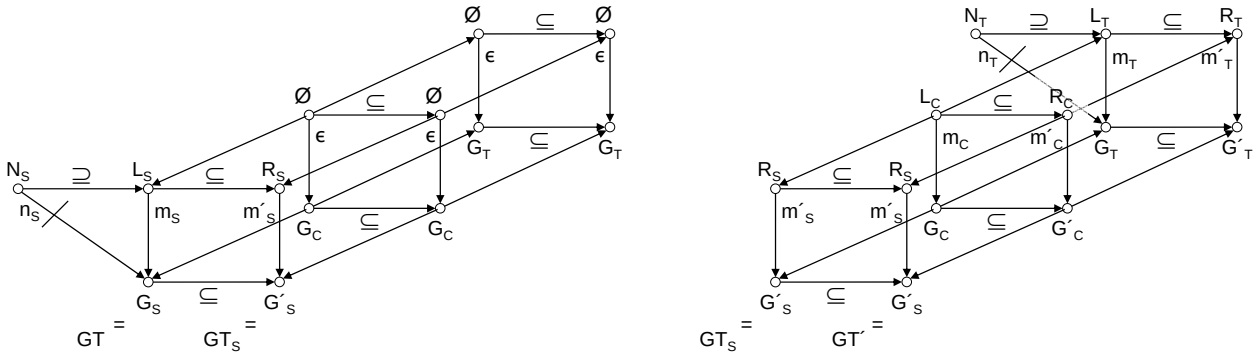


Figure 5.5.: Splitting of Production Triple Application into r_S and r_{ST}.

The application of the *source-to-target domain translating production triple*, i.e., *forward graph translation rule* r_{ST} (cf. right-hand side of Fig. 5.5) keeps the source graph unmodified but adjusts the correspondence and target graph as follows: the effect of applying first r_S and then r_{ST} to a given graph triple is the same as applying p itself if (and only if) we keep the source domain redex, i.e., the morphism m'_S, fixed. Figure 5.5 depicts the two rewriting steps $GT \overset{r_S}{\rightsquigarrow} GT_S \overset{r_{ST}}{\rightsquigarrow} GT'$. These two rewriting steps are equal to $GT \overset{p}{\rightsquigarrow} GT'$, i.e., the combination of left-hand and right-hand side of Fig. 5.5 is equal to Fig. 5.3.

Thanks to Theorem 1 the source component of r_{ST} does not have to check any NACs on the source graph as long as any regarded source graph does not violate any graph constraints, i.e., as long as it has been constructed by means of integrity-preserving productions only. As a consequence, we need no longer care about positive/negative rule application conflicts on the source side when translating a source graph into a related target graph.

Definition 19. *Forward Graph Translation Rules.*
With p being constructed as listed above in Def. 18 the derived forward graph translation rule (FGT rule) *is* $r_{ST} := (p^-_{S,id} \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$ *with components:*
(1) $p^-_{S,id} := (R_S, R_S, \emptyset)$, *i.e., the source component* p_S *of* p *without any NACs that matches and preserves the required subgraph of the source graph only*
(2) $p_C := (L_C, R_C, \emptyset)$, *i.e., the unmodified correspondence component of* p
(3) $p_T := (L_T, R_T, \mathcal{N}_T)$, *i.e., the unmodified target component of* p

For a detailed definition of r_{ST} that includes the morphisms between its rule components as well as for the definition of r_S the reader is referred to [Sch94]. The definitions presented there can be adapted easily to the scenario of integrity-preserving graph triple rewriting as done here for the case of FGT rules r_{ST}.

The definition of a *backward graph translation rule* (BGT rule) r_{TS} is as follows.

Definition 20. *Backward Graph Translation Rules.*
With p being constructed as listed above in Def. 18 the derived backward graph translation

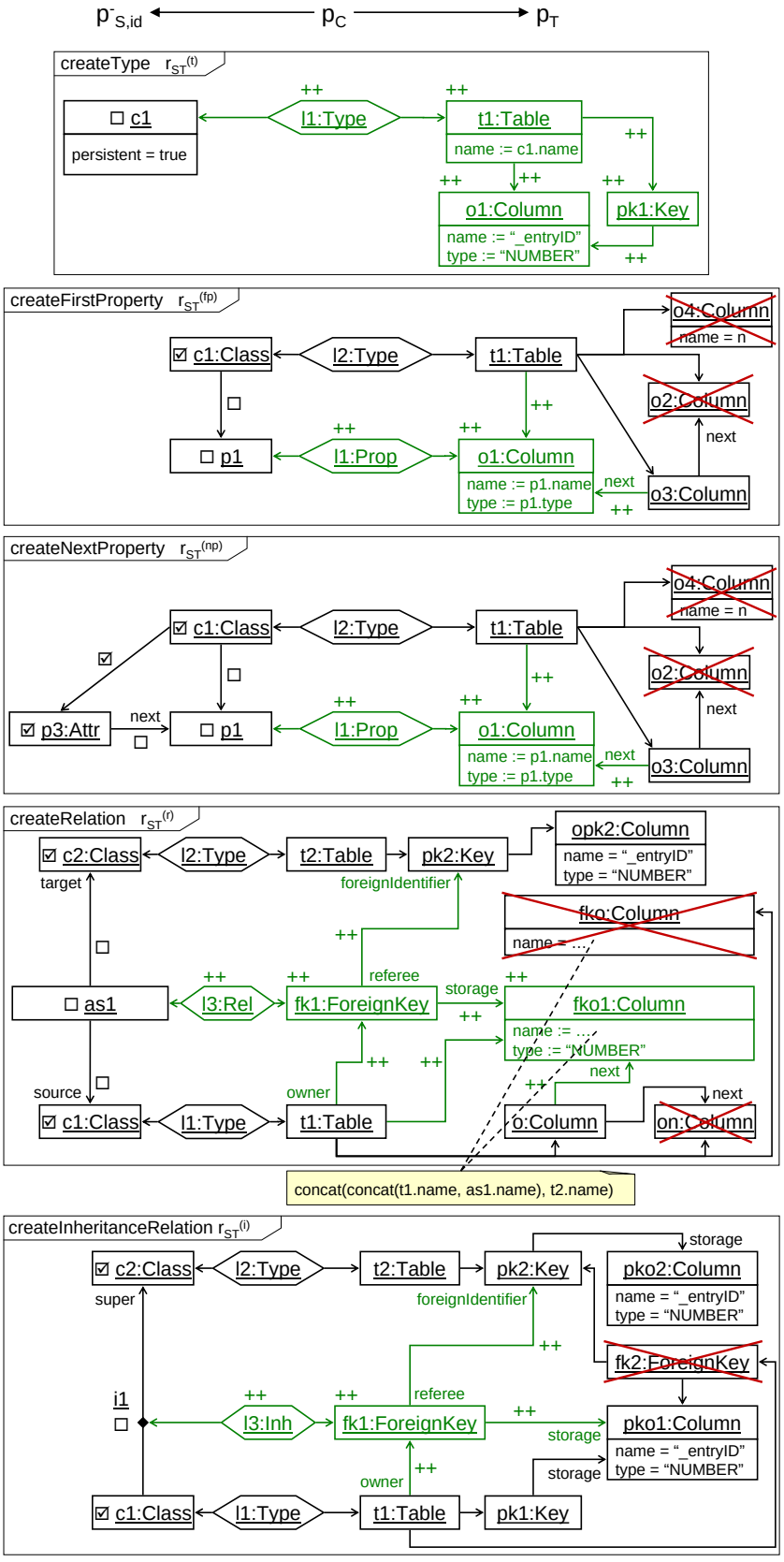


Figure 5.6.: Forward translation rules r_{ST} derived from TGG_{CDDS}.

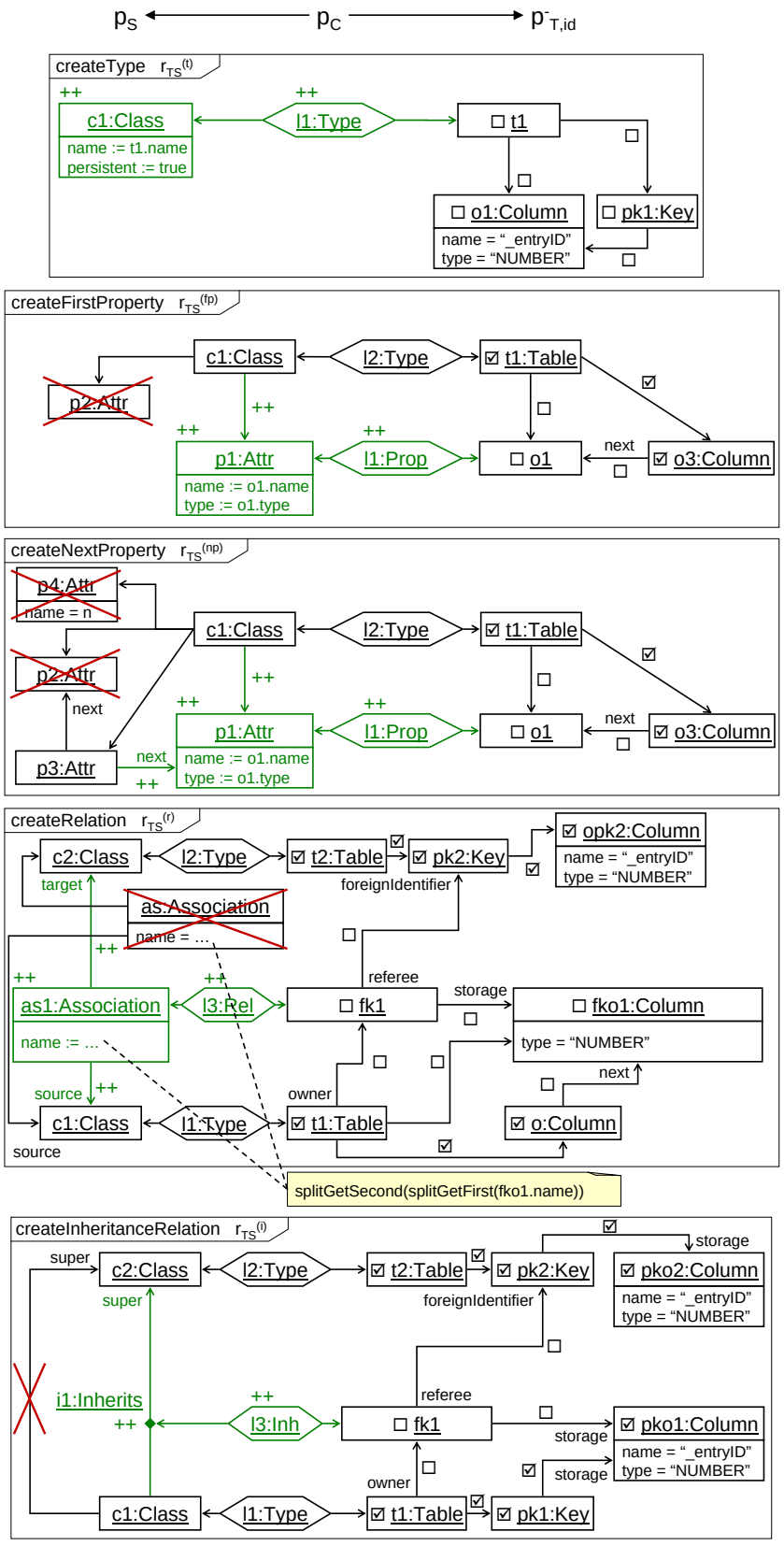


Figure 5.7.: Backward translation rules r_{TS} derived from TGG_{CDDS}.

rule (BGT rule) *is* $r_{TS} := (p_S \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p^-_{T,id})$ *with components:*
(1) $p_S := (L_S, R_S, \mathcal{N}_S)$*, i.e., the unmodified source component of* p
(2) $p_C := (L_C, R_C, \emptyset)$*, i.e., the unmodified correspondence component of* p
(3) $p^-_{T,id} := (R_T, R_T, \emptyset)$*, i.e., the target component* p_T *of* p *without any NACs that matches and preserves the required subgraph of the target graph only*

Figures 5.6 and 5.7 depict the forward translation rules r_{ST} and backward translation rules r_{TS} respectively that have been derived from the productions of TGG_{CDDS} (cf. Fig. 4.6). The derivation process of forward/backward translation rules from a TGG production involves the following steps:

- The primary element of the input component is set to "*bound*" (cf. Sect. 2.8.3). That is, a primary element contained in the input graph component of a graph triple is given as parameter to the translation rule.
- NACs are removed from the input component.
- Elements contained in the input component that are marked as «create» are marked as to-be-translated by an empty checkbox. The «create» annotation is then removed.
- «context» elements in the input component are marked as have-to-be-translated-already by a marked checkbox.
- Attributes with TGG parameter: the parameter in the output domain is replaced by the corresponding value of the input domain. The attribute in the input domain is removed.
- Remaining attributes in the input component that are set to ":=" are set to "=".

When the backward translation rule $r^{(r)}_{TS}$ is derived from TGG production "createRelation" the operation "concat" has to be treated in a special way. Concat's inverse operations "splitGetFirst" and "splitGetSecond" have to be used when calculating the attribute value of association's $as1$ attribute *name* in the source domain using the name of column $fko1$ of the target domain. The value is extracted from the concatenated string stored in *fko1.name* by the following call: splitGetSecond(splitGetFirst(fko1.name))

The implementation of operation "splitGetFirst" searches for the last "concatenation point", i.e., the last occurrence of the reserved character '@'. It then returns the left-hand side string, i.e., the string before the concatenation point. Let us assume *fko1.name* has been set to the value "Author@Writes@Publication". Then "splitGetFirst" returns "Author@Writes". Finally, operation "splitGetSecond" searches for the first "concatenation point", i.e., the first occurrence of character '@'. Therefore, if "Author@Writes" is given to operation "splitGetSecond" it returns the string after the first concatenation point, i.e., "Writes". Consequently, the name of assocation $as1$ is set to "Writes".

splitGetSecond(splitGetFirst("Author@Writes@Publication")) =
splitGetSecond("Author@Writes")) = "Writes"

It is important that the names of classes and tables as well as parameter n do not contain the reserved character '@'. Otherwise "splitGetFirst" and "splitGetSecond" would return wrong

values. Imagine the source table has been wrongly named "Famous@Author" and the target table "Famous@Publication". The TGG parameter n given to the local rule was "Writes". The resulting name of the foreign key column is "Famous@Author@Writes@Famous@Publication". During backward translation the name of association $as1$ would be calculated as follows:

splitGetSecond(splitGetFirst("Famous@Author@Writes@Famous@Publication")) =
splitGetSecond("Famous@Author@Writes@Famous") = "Author@Writes@Famous"

That is, the name would be wrongly set to "Author@Writes@Famous" instead of "Writes".

5.2.4. Local Completeness Criterion

Definition 21 introduces the so-called *local completeness criterion* of the source domain which must be satisfied by the productions of a TGG. Essentially the definition requires that any sequence $SEQ^n_{i=1}(r_{S,i})$ of source-local rules can be completed to a sequence $SEQ^n_{i=1}(r_{ST,i})$ of derivation steps of a graph triple GT that exactly mimics the derivation of its source graph G_S. This criterion will be used later on in Sect. 6.8 to prove the completeness of the introduced algorithm that translates a given source graph G_S into a compatible target graph G_T together with a graph G_C that connects G_S and G_T appropriately.

Definition 21. *Source-Local Completeness Criterion.*
A triple graph grammar TGG fulfills the source-local completeness criterion *iff*
for all $GT_i := (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ *and*
$p := (p_S \leftarrow p_C \rightarrow p_T) \in \mathcal{P}$ *with* $G_S \overset{p_S@m_S}{\rightsquigarrow} G'_S$
exists $p^* := (p^*_S \leftarrow p^*_C \rightarrow p^*_T) \in \mathcal{P}, \quad m^* := (m^*_S, m^*_C, m^*_T)$, *and*
$GT^*_{i+1} := (G'_S \leftarrow G'_C \rightarrow G'_T) \in \mathcal{L}(TGG)$
such that $GT_i \overset{p^*@m^*}{\rightsquigarrow} GT^*_{i+1}$

Definition 22 defines a similar criterion of the target domain.

Definition 22. *Target-Local Completeness Criterion.*
A triple graph grammar TGG fulfills the target-local completeness criterion *iff*
for all $GT_i := (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ *and*
$p := (p_S \leftarrow p_C \rightarrow p_T) \in \mathcal{P}$ *with* $G_T \overset{p_T@m_T}{\rightsquigarrow} G'_T$
exists $p^* := (p^*_S \leftarrow p^*_C \rightarrow p^*_T) \in \mathcal{P}, \quad m^* := (m^*_S, m^*_C, m^*_T)$, *and*
$GT^*_{i+1} := (G'_S \leftarrow G'_C \rightarrow G'_T) \in \mathcal{L}(TGG)$
such that $GT_i \overset{p^*@m^*}{\rightsquigarrow} GT^*_{i+1}$

The local completeness criteria demand that for each local graph (G_S or G_T) of all graph triples $GT \in \mathcal{L}(TGG)$, which is rewritten by the local component of a production p (into G'_S or G'_T), there must be at least one production p^* (p^* may equal p) which rewrites the graph triple GT into GT^*. Therefore, each match $m'_I(R_I \backslash L_I)$ of an input component $p^-_{I,id}$ of a translation rule r_{IO} that identifies not yet translated elements in an input graph can be completed to a full match on the correspondence and output graphs. This is due to the fact that at least one local rule r_I (derived from a production p) exists that has created the matched yet untranslated elements in the input graph. According to the local completeness

criterion a production p^* exists from which a local rule r_I^* is derived that creates the same elements as r_I. That is, the to-be-created elements of p and p^* are equal. For example, the to-be-created elements in the source components of TGG productions "createFirstProperty" and "createNextProperty" are equal. Hence, a translation rule r_{IO}^* exists that has an equivalent input component to r_{IO} which is able to translate the matched not yet translated elements.

In order that productions of TGG_{CDDS} satisfy both local completeness criteria of the source and target domain, the following OCL invariants have to be added to the TGG schema.

$inv_{CD:CD:reservedChar}$ Reserved character '@' must not be used in the name of any classdiagram element, i.e., class, association, and attribute.
context CDElement inv: name.indexOf("@") = 0

$inv_{DS:T:reservedChar}$ Reserved character '@' must not be used in the name of any table.
context Table inv: name.indexOf("@") = 0

$inv_{CD:A:reservedName}$ The reserved name "_entryID" must not be used as attribute name.
context Attr inv: name <> "_entryID"

Now, TGG_{CDDS} satisfies the local completeness criteria. The local completeness criteria demand that when a local rule (r_S or r_T) is applied to an input graph (G_S or G_T) then at least one TGG production p^* must exist that rewrites the complete graph triple GT. Therefore, the local completeness criteria do not allow to define TGG productions that make restrictions in one domain which is not present in the other domain. That is, they enforce a one-to-one translation. In general, symmetric productions satisfy the local completeness criteria.

- TGG production "createType" satisfies the criteria because this production is always applicable and its application is not blocked by any negative application condition.

- If the source component of "createFirstProperty" rewrites the source graph of a valid graph triple then the target component will always find a last column to which it may add the newly created column. The NAC in the target domain that checks whether there exists a column with the same name will never block because any yet existing column will either have the name "_entryID", which is forbidden as attribute name, or will contain two '@' characters due to the name of a column that belongs to a foreign key relation.

  If the source component of "createNextProperty" rewrites the source graph then the target and correspondence graph are rewritten too because the TGG production is symmetric.

  The target components of TGG productions "createFirstProperty" and "createNextProperty" are equal. If one of these productions rewrites a target graph of a valid graph triple, the source and correspondence components will always be able to rewrite the source and correspondence graphs because one of these productions is always applicable: either "createFirstProperty" if the first attribute is created or "createNextProperty" if at least one attribute already exists in the context class.

- If the source component of TGG production "createRelation" rewrites a source graph then the newly created association needs to have a unique name. The NAC in the target domain that ensures that the target graph contains only relations with unique names would only block if an association with the same name was created earlier (which is not the case because the NAC in the source domain did not block).

  The target component of TGG production "createRelation" adds a foreign key and column to the target graph when rewriting a target graph. The name of the relation must be unique in the target graph and, therefore, will be unique in the source graph which is always rewritable because there are no further restrictions in the source domain.

- TGG productions "createRelation" and "createInheritanceRelation" fulfill the local completeness criteria because they have symmetric source and target components.

5.2.5. Conclusion

We have introduced a class of NACs that allow us to derive translators that satisfy the correctness and completeness properties. This class of NACs is used in integrity-preserving TGG productions to generate graphs that fulfill certain graph constraints. As a consequence, derived translation rules are complete, i.e., they can be used to translate any given input graph of a TGG language into a properly related graph. Furthermore, Theorem 1 guarantees the correctness of derived translation rules even if NACs are omitted. Consequently, derived rules never translate input graphs of a TGG language into output graphs such that the resulting graph triple is not an element of the just regarded TGG language.

Due to these achievements we are able to build translators that are correct and complete with respect to their TGG. During the translation process a translator parses a given input graph in order to find a valid sequence of translation rules r_{ST} and r_{TS} that mimics the derivation of the input graph. Although the TGG productions contain NACs these can be safely ignored in the parsing process in the case of integrity-preserving productions. Therefore, positive/negative rule application conflicts are prevented on the input graph. Positive/negative conflicts on the output graph will not lead to dead-ends during parsing because the local completeness criterion guarantees that for each remaining untranslated element in the input graph, created by a local rule, a translation rule exists that is able to translate these elements.

Unfortunately, we still have to solve one problem: in general we are only able to guarantee the completeness of a derived graph translator if we explore an exponential number of derivation paths (w.r.t. the size of a given input graph) due to the remaining positive/positive rule application conflicts. The following section 5.3 will solve this efficiency problem for a sufficiently large class of TGGs (from a practical point of view) by introducing a new application condition for translation rules. This condition rules out any situation, where more than one rule can be used to translate a just regarded node of the input domain in a related subgraph of the output domain.

5.3. Dangling Edge Condition (DEC)

Translators derived from a TGG face certain difficulties concerning the selection of an appropriate sequence of translation rules in the presence of positive/positive rule application conflicts. Therefore, we have introduced a mechanism named *dangling edge condition* in [KLKS10] that checks whether there are incident edges to the node that is currently translated before a translation rule is applied. If some incident edge remains untranslated and is not translatable later on the translation rule is not applied. In the subsequent sections we will discuss the dangling edge condition.

5.3.1. Motivation

Reconsider our triple graph grammar TGG_{CDDS} from Sect. 4.4. An FGT derived from TGG_{CDDS} translates class diagrams to database schemata.

Figure 5.8 (a) depicts a graph that consists of one class c and two attributes $a1$ and $a2$. The empty checkboxes denote that the elements next to them are not yet translated. The graph is valid, as it is derivable by applying TGG productions $p^{(t)}$ and afterwards $p^{(fp)}$ and $p^{(np)}$ of TGG_{CDDS} (cf. Fig. 4.6 in Sect. 4.4) to the empty graph triple. The graph is given as input graph to the FGT. First, the translator applies FGT rule $r_{ST}^{(t)}$ derived from TGG production $p^{(t)}$, which translates class c. Next, rule $r_{ST}^{(fp)}$ is applicable in the context of class c that translates attribute $a1$ (cf. Fig. 5.8 (b)). Finally, both rules $r_{ST}^{(fp)}$ and $r_{ST}^{(np)}$ are applicable in the context of class c that are able to translate attribute $a2$. If the translator chooses to translate attribute $a2$ via rule $r_{ST}^{(fp)}$ the source graph would contain two translated attributes $a1$ and $a2$ with an untranslated link[2] between them (cf. Fig. 5.8 (c)). Unfortunately, no rule exists that is able to translate the remaining untranslated edges.

So, the translator produced a so-called *dangling edge* in the source graph. Consequently, the translator states at the end of the translation process that it is not able to translate the (valid) input graph completely due to the *dangling edge*. Alternatively, the translator could perform backtracking to the decision point in Fig. 5.8 (b) and apply rule $r_{ST}^{(np)}$ instead of $r_{ST}^{(fp)}$.

Whenever constellations in the input graph appear, where two or more rules are applicable that translate overlapping sets of input graph elements, translation algorithms are demanding for help to select the appropriate rule. We propose an extension that is inspired by building parsers for compilers and related techniques for parsing words that are passed to the compiler. Typically, top-down and bottom-up parsers decide on more information than just the recent input: they take a *look-ahead* into account. In the following we introduce the so-called *dangling edge condition* (DEC) that prevents the application of a rule if the rule would produce a dangling edge. TGG translators produce dangling edges if an edge is still untranslated at the end of the translation process. So, translators must ensure that before applying a rule another translation rule exists that is able to translate this currently "dangling" edge later on. This DEC is inspired by an analogous condition in DPO approaches, which explicitly prohibits deleting a node without deleting all incident context edges as part of the same rule application step. This way, our DEC eliminates positive/positive rule application conflicts.

[2]Note that a link consists of 3 nodes and 4 edges in our approach that maps models to graphs (cf. Sect. 2.8.1).

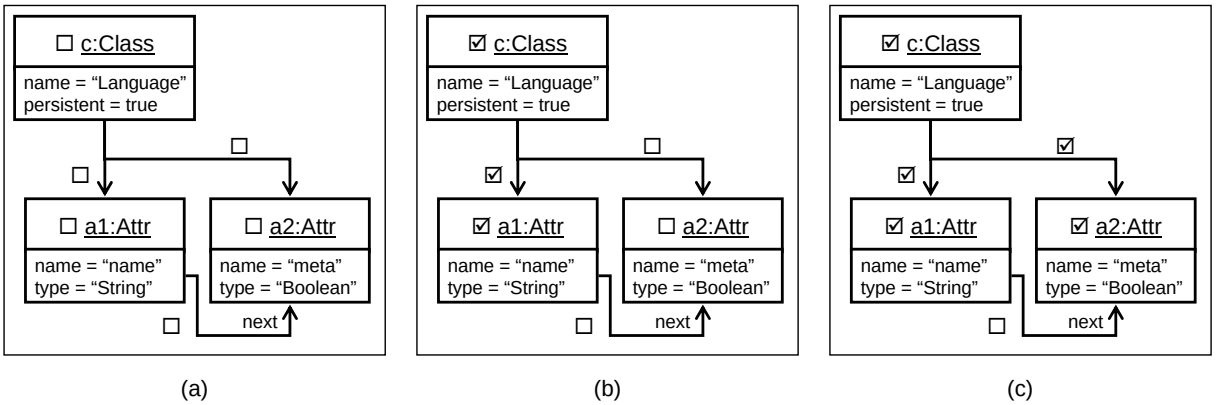


Figure 5.8.: (a) Input graph given to the FGT, (b) Input graph partially translated, (c) Input graph translated with dangling edges

We restrict our focus to forward translators in the sequel, but all concepts and ideas can be transferred to backward translators as well.

The core idea of the DEC is that several productions may be applicable such that their matches overlap in some node. If the production with the smaller match is applied, incident edges cannot be translated later on. The DEC resolves conflicts where context-sensitive productions create one primary node that is connected via new edges to at least one context node. It does not offer a solution for those cases where the created nodes are not connected.

In the following, we regard TGG productions that create only one primary node on each side. Primary nodes of context-sensitive productions must be connected to at least one context node. Secondary nodes must be either connected to a primary node or another secondary node. The graphs that result by applying such productions are either graph structures that are not connected to other structures (in case of applying initial context-free productions like production $p^{(t)}$ of TGG_{CDDS}) or connected graph structures (in case of applying context-sensitive productions $p^{(fp)}$, $p^{(np)}$, $p^{(r)}$, and $p^{(i)}$ of TGG_{CDDS}).

5.3.2. Formal introduction to LNCC and DEC

As shown in Sect. 5.3.1, application of certain translation rules may lead to invalid graph triples since some edges in the graph of the input domain remain untranslated. Based on this observation we define for the source graph of a TGG the so-called *Legal Node Creation Context* relation with a look-ahead of one $LNCC_S(1)$ that will be used to control the selection and application of FGT rules. A relation $LNCC_T(1)$ used by BGTs is constructed similarly. TGG productions can be broken down to certain fragments, where at most two nodes make up a part of the production. Elements of $LNCC_S(1)$ are *4-tuples* that represent certain kinds of source graph production fragments. The first and third component of a tuple represent the type of the node that is the source and target of an edge e created by a production respectively. The type of this edge e is used as second component. The fourth component denotes whether the source node, target node, or both nodes are used as context in the production fragment. Tuples of $LNCC_S(1)$ are derived from a given TGG as follows:

Definition 23. *Legal Node Creation Context with a look-ahead of 1.*
$LNCC_S(1) \subseteq V_{TG_S} \times E_{TG_S} \times V_{TG_S} \times \{s,t,st\}$ *is the smallest* legal node creation context relation *for the source graph of a given TGG such that*
$(vt_s, et, vt_t, c) \in LNCC_S(1)$ *iff*
(1) $\exists$ *TGG production* $((L_S, R_S, \mathcal{N}_S) \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$ *that creates edge* $e \in R_S \backslash L_S$ *with at least one incident already existing context node* $s(e)$ *or* $t(e) \in L_S$
(2) $vt_s = type(s(e))$, *(3)* $et = type(e)$, *(4)* $vt_t = type(t(e))$
(5) $c \in \{s,t,st\}$, *with the following semantics:*
(5.1) s: $s(e) \in L_S$, $t(e) \in R_S \backslash L_S$
(5.2) t: $t(e) \in L_S$, $s(e) \in R_S \backslash L_S$
(5.3) st: $s(e), t(e) \in L_S$

Figures 5.9 (a), (b), and (c) identify all possible node and edge constellations that contribute tuples to $LNCC_S(1)$. In addition, Figs. 5.9 (d), (e), and (f) depict those production fragments that do not contribute any tuples to $LNCC_S(1)$.

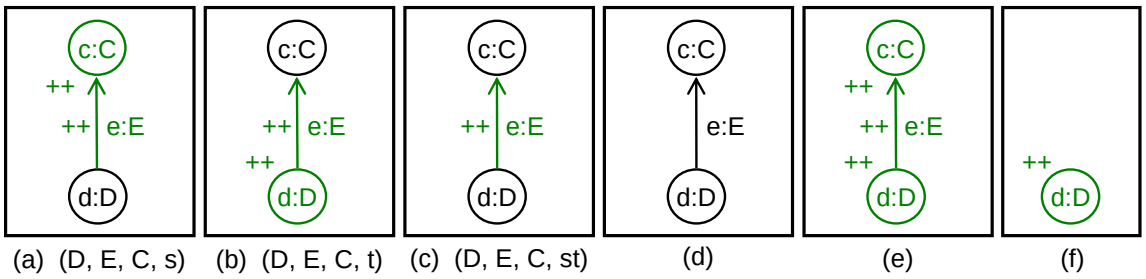


Figure 5.9.: TGG production fragments relevant and irrelevant for $LNCC_S(1)$.

The motivation behind the definition of $LNCC_S(1)$ is to block a translation of a node of the source graph that has incident edges that are not translated in the same step and that cannot be translated later on (i.e., to avoid dangling edges). This situation occurs if a TGG contains overlapping productions (e.g., productions $p^{(fp)}$ and $p^{(np)}$ of TGG_{CDDS}). These productions are applicable in the same context and create a node of the same type (both $p^{(fp)}$ and $p^{(np)}$ create nodes of type "Attr") but at least one production creates an edge that relates the new node to an already existing node ($p^{(np)}$ creates a link from the new attribute to its predecessing attribute). Therefore, a translator that applies one of the rules derived from these productions would destroy the match of the other rule and potentially leave an untranslatable edge. In order to identify such dangling edge situations, TGG production fragments must be inspected which create edges where the source or the target of the edge already exists, i.e., is used as context (cf. Figs. 5.9 (a), (b), and (c)). Translation rules derived from TGG productions containing these fragments have the potential to translate edges of the input graph using one or two already translated incident nodes as context. As patterns (d) to (f) do not translate such edges they can be neglected. Pattern (a) depicts a production fragment in which node d is the already existing context for the new node c and d is the source of the new edge e ($s(e) = d$). In production fragment (b) node c is used as context and c is the target of edge e ($t(e) = c$). Pattern (c) depicts a situation where a new edge between nodes c and d is created, i.e., both nodes are used as context.

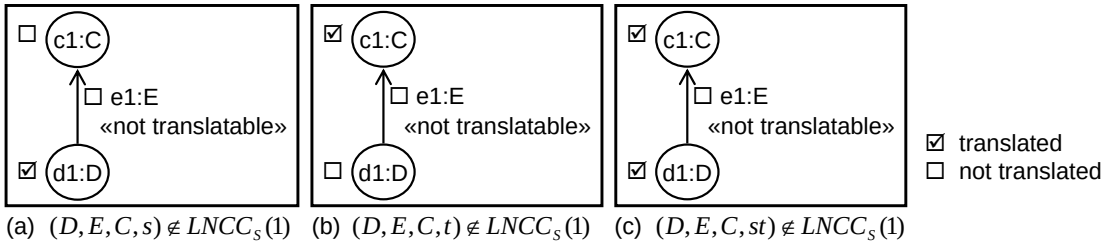


Figure 5.10.: Patterns in input graph that violate DEC(1).

Whenever we encounter a not translated edge with an already translated incident node, we will use the relation $LNCC_S(1)$ to check whether an FGT rule exists that can be used later on to translate the regarded edge. If $LNCC_S(1)$ does not contain an appropriate tuple then the just regarded edge cannot be translated. On the other hand, the existence of an appropriate tuple does not guarantee that the edge is translatable. This is due to the fact that FGT rules r_{ST} containing a tuple are only applicable if a match of a rule's complete left-hand side $(R_S \leftarrow L_C \rightarrow L_T)$ is found in the host graph triple and no NAC in the target domain blocks. In general we have to restrict the application of translation rules such that the situations depicted in Fig. 5.10 are avoided:

(a) Node $c1$ is not translated yet but $d1$ (the source of $e1$) is and there exists no rule with production fragment (D, E, C, s) that may translate $e1$ later on.
(b) Node $d1$ is not translated yet but $c1$ (the target of $e1$) is and there exists no rule with production fragment (D, E, C, t) that may translate $e1$ later on.
(c) Nodes $d1$ (source) and $c1$ (target) are both translated and there exists no rule with production fragment (D, E, C, st) that may translate $e1$ later on.

Therefore, the application of a translation rule must satisfy certain application conditions given in Def. 24 including the *Dangling Edge Condition* $(DEC(1))$.

Definition 24. *Rule application conditions for FGTs with a look-ahead of 1.*
Let TX be the set of already translated elements of the source graph G_S, $e \in E_S$, and p be a TGG production $((L_S, R_S, \mathcal{N}_S) \xleftarrow{h_S} p_C \xrightarrow{h_T} p_T)$. Thus, for each match m'_S of translation rule r_{ST} in G_S the rule application conditions (1) to (3) must hold including the dangling edge condition *$DEC(1)$ that consists of the subconditions $DEC_1(1)$, $DEC_2(1)$, and $DEC_3(1)$ in order to apply r_{ST} to $(G_S \leftarrow \ldots \rightarrow \ldots)$:*

(1) $m'_S(L_S) \subseteq TX$ *(context elements are already translated)*
(2) $\forall x \in m'_S(R_S \backslash L_S) : x \notin TX$ *(no element x shall be translated twice)*
(3) $TX' := TX \cup m'_S(R_S \backslash L_S)$ *(TX is extended with translated elements)*

($DEC_1(1)$) $\forall e \notin TX'$ *where* $s(e) \in TX', t(e) \notin TX'$:
$(type(s(e)), type(e), type(t(e)), s) \in LNCC_S(1)$

($DEC_2(1)$) $\forall e \notin TX'$ *where* $t(e) \in TX', s(e) \notin TX'$:
$(type(s(e)), type(e), type(t(e)), t) \in LNCC_S(1)$

($DEC_3(1)$) $\forall e \notin TX'$ *where* $s(e), t(e) \in TX'$:
$(type(s(e)), type(e), type(t(e)), st) \in LNCC_S(1)$

Def. 24 thus introduces a rather straightforward way to decide if a translation rule shall

be applied or not just by looking at the 1-context of a to-be-translated node. By adding this condition to the translation algorithm defined in [SK08] (cf. Listing 4 in Sect. 6.6), we are able to reduce the number of situations significantly, where we were forced to choose one of the applicable rules nondeterministically and run into dead-ends due to the wrong choice. In general, Def. 24 is not able to resolve all positive/positive conflicts, i.e., there may be multiple rules that are able to translate a node using different matches, i.e., matches containing different to-be-translated elements. Therefore, algorithm in Listing 4 will abort in this case. Alternatively, the user could be asked which of these elements should be translated or rule priorities [Kön09] can be used to reduce the number of different matches if more than one rule is applicable by filtering matches of rules with low priority.

Though, the algorithm permits multiple *locally-applicable rules*, i.e., rules that translate the same elements. A locally-applicable rule is either applicable also on the whole graph triple or its application is prevented, e.g., due to NACs in the output component.

The set of productions of TGG_{CDDS} contains multiple locally-applicable rules. BGT rules (fp) and (np) are both applicable in the context of the non-first column of a table. These rules are *disjoint applicable*, i.e., only one of the locally-applicable rules is applicable on the whole graph triple (cf. translation example in Sect. 6.7). In general, multiple locally-applicable rules need not to be disjoint applicable because they translate the same elements. Executing one of the locally-applicable rules nondeterministically does not lead into dead-ends due to the local completeness criterion and the same reason why positive/negative conflicts on the target side do not lead into dead-ends (cf. Def. 21 and subsequent discussion).

5.3.3. Extracting LNCC from TGG productions

Given a set of TGG productions we have to extract the LNCCs of the source and target domain. Unfortunately, the discussed TGG productions of our running example TGG_{CDDS} (cf. Fig. 4.6 in Sect. 4.4) contain links and objects rather than edges and nodes (cf. Sect. 4.3). So, we first have to convert the TGG productions of TGG_{CDDS} into a graph representation by using the mapping from models to graphs discussed in Sect. 2.8.1. Then we are able to extract LNCC entries from these graphs.

Figure 5.11 depicts three TGG production fragments that use the link and object representation of TGG elements (cf. left-hand sides of Figs. 5.11 (a), (b), and (c)) and their representation as graph using the mapping defined in Sect. 2.8.1 (cf. right-hand sides of Figs. 5.11 (a), (b), and (c)). The left-hand sides of Fig. 5.11 depict all TGG production fragments that contribute tuples to LNCC. A triangle next to a link indicates which of the ends is the first and second end respectively. The links are translated into a representation consisting of one link node e, two slot nodes $s1$ and $s2$, and four edges. Edges $et1$ and $et2$ that connect the slot nodes with the linked objects are of relevancy. According to the mapping definition these edges and the slot nodes are typed. This results in unique combinations of node and edge types after mapping linked objects from the modeling domain to the graph domain.

We will informally introduce the so-called *Legal Object Creation Context* (LOCC) that refers to the higher level constructs object and link of a TGG production fragment. Entries of a LOCC are more human readable than the resulting LNCC tuples and are used in this contribution when discussing DEC examples. A LOCC is very similar to a LNCC. Each entry

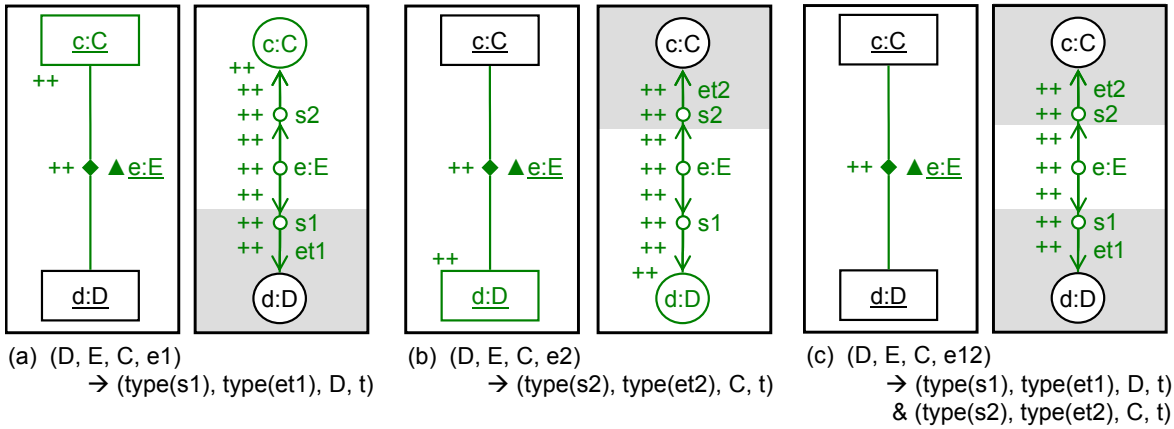


Figure 5.11.: Extracting LNCC from TGG production fragments.

is a 4-tuple. The first and third component of a tuple represent the type of an object, i.e., a class. The second component represents the link type, i.e., association, that connects these two objects. The first component is the first end of the link, whereas the third component is the second end of the link. The fourth component denotes whether the first and/or second end of the link is used as context in the production fragment. It may contain one of the following values: $\{e1, e2, e12\}$ depending on whether the first, the second or both ends of a link are used as context.

The LOCC entry extracted from Fig. 5.11 (a) is $(D, E, C, e1)$ which means that a link is created between objects of types C and D, where D is used as context and is the first end of the link and C the second end. The LNCC tuple derived from this LOCC entry is the fragment containing the type of slot node $s1$, the type of node d and the type of edge $et1$ which connects both nodes: $(type(s1), type(et1), D, t)$. The LOCC entries extracted from Figs. 5.11 (b) and (c) are $(D, E, C, e2)$ and $(D, E, C, e12)$. The LNCC tuples derived from these LOCC entries are $(type(s2), type(et2), C, t)$ and $(type(s1), type(et1), D, t)$ & $(type(s2), type(et2), C, t)$ respectively.

All LOCC entries of source and target domain that are extractable from TGG_{CDDS} are depicted in Table 5.1. These entries can be translated into their LNCC representation using the mapping defined in Fig. 5.11. For example, the LOCC entry extracted from the source component of TGG production "createFirstProperty" is $(Class, Contains, Attr, e1)$. The LNCC entry derived from this LOCC entry is $(type(Contains{::}s1), type(Contains{::}et1), Class, t)$.

(Class, Contains, Attr, e1)
(Attr, Precedes, Attr, e1)
(Association, StartsAt, Class, e2)
(Association, EndsAt, Class, e2)
(Class, Inherits, Class, e12)

$LOCC_S$

(Table, Contains, Column, e1)
(Column, Precedes, Column, e1)
(ForeignKey, RefersTo, Key, e2)
(Table, RelatesToForeignEntriesVia, ForeignKey, e1)
(ForeignKey, StoresForeignKeysIn, Column, e2)

$LOCC_T$

Table 5.1.: LOCC of source and target domain extracted from TGG_{CDDS}.

5.3.4. Dangling Edge Condition by Example

Now, we show by example that checking for dangling edges helps deciding which rule should be applied by translators derived from a TGG if multiple rules are applicable at overlapping matches. Therefore, we consider again the FGT derived from TGG_{CDDS} and the input graph depicted in Fig. 5.8 (b) which has been discussed in Sect. 5.3.1.

As we have already shown, both translation rules (fp) and (np) are applicable after applying rules (t) and (fp) to this input graph. Based on the classification scheme of Fig. 5.11 we construct the set of tuples from the TGG productions of TGG_{CDDS} which results in $LOCC_S(1)$ shown in Table 5.1. Next, we pretend to apply rule (fp) in the context of attribute *a2*. Then, we calculate the set of incident edges of $a2$ that are not yet translated. We must check whether all of these edges are translatable by further rewriting steps, i.e., whether $DEC(1)$ is satisfied. There is one untranslated edge which belongs to the link between $a1$ and $a2$: the "next" link. As both first and second end of this link are already translated, the tuple $(Attr, Precedes, Attr, e12)$ must be in $LOCC_S(1)$ which is not the case. As a consequence, we do not apply FGT rule (fp), because this would result in a dangling edge (cf. Fig. 5.8 (c)) and proceed pretending to apply rule (np). In this case the set of incident edges of $a2$ that are not yet translated is empty. So, the rule application conditions given in Def. 24 are satisfied, i.e., there are no dangling edges. Concluding, we were able to translate the input graph completely due to the fact that the DEC prohibited selecting a wrong translation rule match.

6. Graph Translators for Extended TGGs

Let link [...] be [...] nodes and edges.

[Sect. 2.8.1, Def. 10]

In this chapter we discuss the graph translation algorithms presented in [SK08] and [KLKS10] that are used to implement forward and backward graph translators. In Sect. 6.1 we discuss a framework for common translation algorithms. Afterwards, Sect. 6.2 introduces so-called core rules which are used by the translation algorithms. Section 6.3 discusses the algorithm from [SK08] which supports NACs but does not guarantee completeness. We present this algorithm to demonstrate the overall idea of a translation algorithm and state examples and shortcomings in Sects. 6.4 and 6.5. Section 6.6 discusses a more advanced algorithm from [KLKS10] that handles extended triple graph grammars (cf. Chap. 5) especially NACs as presented in Sect. 5.2 and implements the dangling edge condition (cf. Sect. 5.3). In addition, the algorithm from [KLKS10] has been extended to handle secondary elements. More translation examples are given in Sect. 6.7. Section 6.8 compares the properties of this algorithm with the requirements stated for bidirectional translators in Sect. 3.5. Finally, an algorithm that checks given graph triples for consistency is discussed in Sect. 6.9.

6.1. Graph Translation Algorithm Framework

In our translation framework each translator implements procedure $evolve : GT_{in} \rightsquigarrow^* GT_{out}$ which simulates the simultaneous evolution of a given graph triple GT_{in} (cf. Listing 1). The input graph triple GT_{in} is either $(G_S \leftarrow G_\emptyset \rightarrow G_\emptyset)$ in case of an FGT or $(G_\emptyset \leftarrow G_\emptyset \rightarrow G_T)$ in case of a BGT[1]. The input graph G_{input} is either G_S or G_T depending on the type of translator (FGT/BGT), whereas the output graph G_{output} is either G_T or G_S. Procedure *evolve* assumes that the underlying TGG is integrity-preserving and that the input graph GT_{in} was produced by a sequence of input-local rules r_I, i.e., $G_{input} \in \mathcal{L}(GG_I)$. *Evolve* is able to cope with situations where the underlying TGG or the input is invalid. It throws errors if it detects an invalid TGG specification and exceptions in case of invalid inputs. A valid translation produces an output graph triple $GT_{out} = (G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$. Resulting graph triples of invalid translations are undefined.

[1] In the general case, when incremental updates are performed with a translator, the correspondence graph and the graph of the opposite domain need not to be empty.

Procedure *evolve* is the main entry point of our translation framework. It first initializes a set of global variables that are then available to all subroutines. Two global sets guarantee that nodes of the input graph are only translated once (set *translatedElements*) and that cycles, which might occur due to recursive calls of translation functions, are broken (set *justRegardedElements*). Subsequently, the input graph is checked for validity. Now, *evolve* invokes function *translate(GraphTriple)*, which translates the given graph triple. Finally, the output graph is checked for validity and whether all elements of the input graph have been translated. If all tests are successful then a valid graph triple has been produced and is returned. Otherwise the set of TGG productions might contain integrity-destroying productions. This is the case if a valid input graph has been translated into an invalid output graph. If this is also not the case then the input graph was not derivable by the set of TGG productions.

```
1  procedure GraphTriple evolve(inputGraphTriple: GraphTriple) {  // GT_in
2    global inputGraph: Graph = Translator.getInputGraph(inputGraphTriple);  // G_input
3    global translatedElements: ElementSet = inputGraph.getTranslatedElements();  // TX
4    global justRegardedElements: ElementSet = ∅;
5
6    inputValid: boolean = inputGraph.verifyConstraints(); // Def.11(1)/(3) satisfied?
7    outputGraphTriple: GraphTriple = translate(inputGraphTriple);  // produce GT_out
8    outputValid:boolean= Translator.getOutputGraph(outputGraphTriple).verifyConstraints();
9
10   translated: boolean = inputGraph.isCompletelyTranslated();
11   if (inputValid && outputValid && translated)
12     return outputGraphTriple; // successfully produced GT_out
13   else if (inputValid && not outputValid) // Def.17(1) or (3) violated!
14     throw TGGContainsIntegrityDestroyingProductionsError(outputGraphTriple, translated);
15   else throw InputGraphNotPartOfDerivableGraphTripleException( // user-error: ...
16     outputGraphTriple, inputValid, outputValid, translated);   // ...G_input ∉ L(GG_I)
17 }
```

Listing 1: Procedure evolve: entry point of translation algorithms.

Procedure *evolve* calls subroutine *translate(GraphTriple)* (cf. Listing 2), which in turn calls procedure *translate(Node)* for all nodes in the input graph G_{input}. Procedure *translate(GraphTriple)* translates all objects and links that are contained in the input graph. This procedure might be overridden by an algorithm implementation if required, e.g., to sort the elements to be translated beforehand. Procedure *translate(GraphTriple)* first translates all objects contained in the given graph triple and afterwards translates remaining untranslated links. Therefore, the nodes that belong to the objects and links are fetched and passed to subroutine *translate(Node)*. This subroutine is implemented by every algorithm and contains the main logic that translates a given graph element utilizing derived operational rules. The framework assumes that this implementation satisfies the correctness and completeness properties. The implementation of procedures *getObjectNodes(Graph):Set<Node>* and *getLinkNodes(Graph):Set<Node>* assumes that objects as well as links are realized by nodes in the underlying graph structure (cf. Sect.2.8.1 for a mapping of models to graphs). Consequently, the algorithm operates on nodes only and does not need to operate on edges.

```
1  procedure GraphTriple translate(graphTriple: GraphTriple) {
2    inputGraph: Graph = Translator.getInputGraph(graphTriple);
3    forall (objectNode ∈ getObjectNodes(inputGraph)) { translate(objectNode); }
4    forall (linkNode ∈ getLinkNodes(inputGraph)) { translate(linkNode); }
5    return graphTriple;
6  }
```

Listing 2: Procedure that translates a graph triple by iterating through all nodes.

6.2. Core Rules

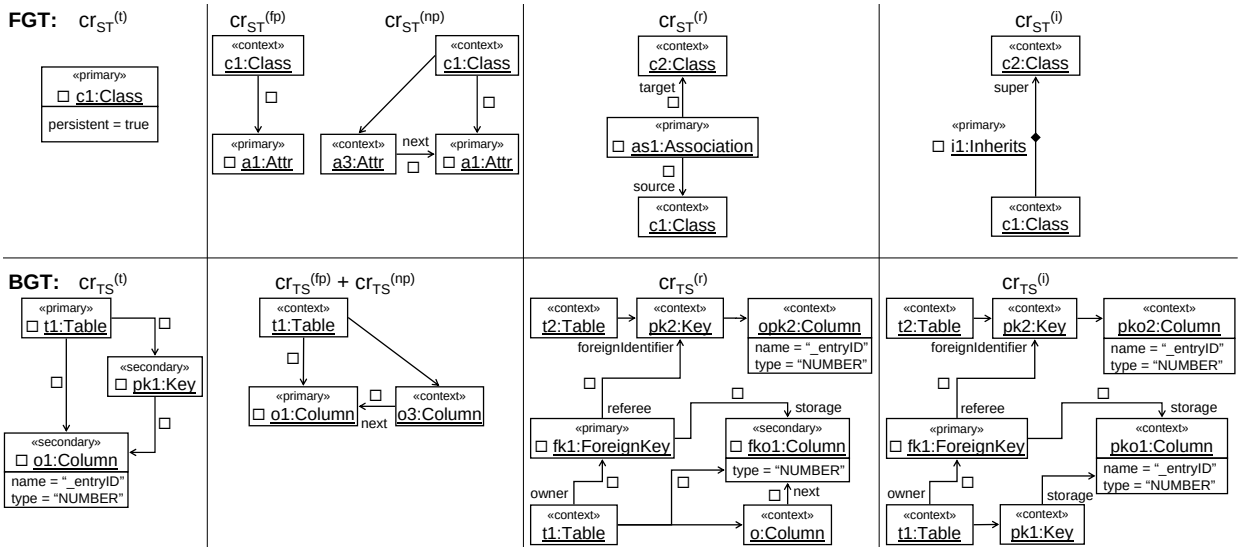


Figure 6.1.: FGT core rules cr_{ST} and BGT core rules cr_{TS} derived from TGG_{CDDS}.

The presented algorithms use so-called *core rules* (cf. [SK08]) to determine *core matches* of translation rules in the input graph and to determine whether a given node is a secondary node. A core rule is closely related to the input component $p^-_{I,id}$ (either $p^-_{S,id}$ or $p^-_{T,id}$) of a translation rule (cf. Def. 19). Fig. 6.1 depicts the core rules cr_{ST} and cr_{TS} derived from TGG_{CDDS}, which are used by forward and backward translators respectively. Two core rules derived from different TGG productions may have an identical core (cf. $cr^{(fp)}_{TS}$ and $cr^{(np)}_{TS}$). In the general case, a core rule may have multiple core matches that consist of the same elements (resulting, e.g., from permutation).

A core rule looks up the context elements of a given primary element in G_{input}, which may or may not be translated already but must be translated before the primary element is translatable (cf. Def. 24 (1)). Therefore, the primary element is given as parameter to the core rule and used as starting point of the pattern matching. The resulting operation is used by algorithms to determine the required context elements (cf. lines 21, 23, and 25 of Listing 4 in Sect. 6.6) and the to-be-translated elements (cf. lines 21, 31, and 44 of Listing 4 in Sect. 6.6), i.e., the primary element and additional secondary elements.

Moreover, a core rule may be used to determine whether a given node is a secondary node and to retrieve the corresponding primary node. This is implemented in the two operations "isSecondaryNode" and "getPrimaryNode" which are required by the algorithm in Listing 4 discussed in Sect. 6.6 (cf. lines 8 and 9). Therefore, the secondary node is given as parameter to the core rule. The operations bind the given secondary node to a type compatible secondary node of a core rule, use this node as starting point of the pattern matching, and search for a full match of the core rule in the host graph. If a match is found, the given node is a secondary node. The associated primary node is uniquely determined by the match and returned by operation "getPrimaryNode". Note that these operations can only work properly

if the matching structure of secondary elements is unique in all specified TGG productions. It is the responsibility of the TGG designer to make secondary elements uniquely determinable. According to the core rules of TGG_{CDDS}, columns or primary key elements in the target domain may be secondary elements.

Core rules contain elements of the input graph only. NACs are not contained in a core rule. The primary and secondary elements and additional incident edges given to a core rule must not be translated yet. This is indicated by the empty checkboxes next to these elements.

6.3. Simple Graph Translation Algorithm

In this section we discuss an algorithm presented in [SK08] that has been embedded into our translation framework and slightly adjusted due to readability. This algorithm is not complete because it might not always find a sequence of translation rules that are able to completely translate every given valid input graph. Moreover, the algorithm translates an element of the input graph non-deterministically if more than one rule is able to translate this element by executing one of the appropriate rules non-deterministically. But, the algorithm is able to cope with NACs in general and is not limited to the class of NACs discussed in Sect. 5.2.1, i.e., NACs that are only used to prevent the creation of graphs which violate constraints in the source or target domain. The translation algorithm from [SK08] is depicted in Listing 3.

Procedure *translate(Node)* immediately returns if the regarded node has been translated already or if the node is just regarded. The latter might happen if the current procedure call is nested in another call to *translate(Node)* due to recursive context translation. If the node is not translated and not regarded right now it is marked as regarded and the algorithm starts searching for a translation rule that is able to translate the node. Therefore, the algorithm calculates a core rule match of any rule that is able to translate nodes of the same type. If a match is found then all nodes that are required as context for the current node are recursively translated. If all required context elements are translated afterwards then the rule is marked as appropriate rule. The algorithm proceeds collecting all appropriate rules. After all appropriate rules have been collected it executes one appropriate translation rule for some match that includes the node to be translated non-deterministically. If the appropriate rule has been executed successfully the application of further appropriate rules is stopped and elements that have been translated are marked as translated. Contrary to the proposed handling of NACs of the input domain—i.e., NACs can be safely removed from the input component of a translation rule derived from an integrity-preserving production (cf. Chap. 5)—NACs of the input domain are not removed from the input domain in this algorithm. That is, translation rules derived from a TGG production (cf. Figs 5.6 and 5.7 in Sect. 5.2.3) contain NACs in their input domain (which is not depicted in the figures). The elements in these NACs have marked checkboxes indicating that these elements might exist in the input graph but must not be translated already. Note that it might be possible that the execution of one rule prevents another rule that contains a NAC from being applied because it creates a structure which is forbidden by this NAC (cf. *positive/negative rule application conflict* in Sect. 5.2). Such a situation might result in an incomplete translation because the appropriate sequence of translation rules cannot be determined by the algorithm in this case. Procedure *evolve* of

the algorithm framework assumes that the translation operation implemented by a translation algorithm always fulfills the completeness property. Consequently, it would wrongly state that the input graph is not part of a graph triple that is an element of the language defined by the TGG because the implementation of procedure *translate(Node)* in Listing 3 does not fulfill the completeness property.

```
1  procedure void translate(n: Node) {
2    if (n ∈ translatedElements or n ∈ justRegardedElements)
3      return;
4    else {
5      justRegardedElements.add(n);
6      candidateRules: RuleSet = select rules where r.primaryInputNode.type equals n.type;
7      forall (rule ∈ candidateRules) {
8        compute core matches of rule in inputGraph with n as primary node;
9        if (at least one core match found) {
10         forall (contextNode ∈ context elements of core match)
11           { translate(contextNode); }
12         if (translatedElements contains all context elements of core match)
13           appropriateRules.add(rule, core match);
14     } }
15     forall ((rule, core match) ∈ appropriateRules) {
16       execute rule for some completed match and extend outputGraph
17         (regarding NACs that must not find a match with translatedElements);
18       if (successfully executed) {
19         translatedElements.add(elements of inputGraph translated by rule);
20         break;
21     } }
22     justRegardedElements.remove(n);
23 } }
```

Listing 3: TGG Algorithm equivalent to algorithm from [SK08].

6.4. Forward Translation Example

In the following we discuss an example of a forward translator derived from TGG_{CDDS} which uses the algorithm in Listing 3. The input graph given to the translator is the class diagram depicted in Fig. 6.2. The class diagram contains two classes that have an inheritance relationship and three ordered attributes. The class diagram was produced by applying patterns "createPersistentClass", "createPersistentClass", "createGeneralization", "createFirstAttribute", "createNextAttribute", and "createNextAttribute" in that order. The algorithm now has to find a sequence of translation rules that mimics the simultaneous evolution of a proper graph triple. An appropriate sequence is, e.g., $(r_{ST}^{(t)}, r_{ST}^{(fp)}, r_{ST}^{(np)}, r_{ST}^{(np)}, r_{ST}^{(t)}, r_{ST}^{(i)})$. The algorithm may start the translation process with any node because it recursively translates the context of the current node before it translates the node itself. Let us assume the translation order in procedure *translate(GraphTriple)* of objects is $(c2, c1, a2, a3, a1)$ and the order of links is $(e2, e6, e3, e5, e4, e1)$. The algorithm framework calls procedure *translate(Node)* passing the according object and link nodes.

The first object which is translated is class $c2$. Translation rule $r_{ST}^{(t)}$ is the only candidate rule. The core rule match contains class $c2$ only. No additional context has to be translated beforehand. Therefore, $r_{ST}^{(t)}$ is marked as appropriate rule and then executed. The output graph and the correspondence graph are extended by a new table, a primary key, a primary

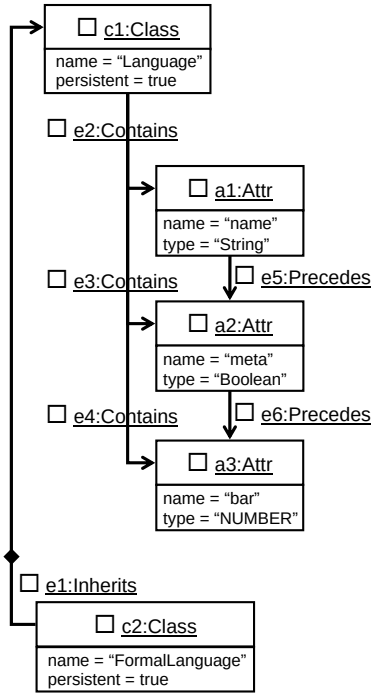


Figure 6.2.: Input graph given to forward translator.

key column, and a TGG link that connects class $c2$ with the newly created table (cf. Fig. 6.3). Class $c2$ is added to the set of translated elements.

Next, class $c1$ is translated. The operating sequence is identical to the translation of class $c2$. So, translation rule $r_{ST}^{(t)}$ is executed which produces another table, additional secondary elements and a TGG link that connects class $c1$ and the table. Class $c1$ is added to the set of translated elements.

Right now, the second attribute $a2$ of class $c1$ is translated. There are two candidate rules: $r_{ST}^{(fp)}$ and $r_{ST}^{(np)}$. The additional context which has to be translated beforehand is class $c1$, attribute $a1$, and contains link $e2$. Class $c1$ is already translated so the translation call *translate(c1)* immediately returns. The call *translate(a1)* will recursively translate attribute $a1$ before attribute $a2$ is translated. The only candidate rule that finds a match in the context of $a1$ is $r_{ST}^{(fp)}$. Candidate rule $r_{ST}^{(np)}$ does not find a match in the context of $a1$ in the input graph. As the owning class $c1$ is already translated, the algorithm proceeds and executes $r_{ST}^{(fp)}$. The NAC in the input graph does not block because no other attribute contained in class $c1$ has been translated already. Therefore, $r_{ST}^{(fp)}$ is successfully executed and a corresponding column is added to the corresponding table. Attribute $a1$ and the contains link $e2$ are added to the set of translated elements.

Now that the context elements $c1$, $a1$, and $e2$ of attribute $a2$ are translated, the algorithm resumes translating $a2$. The appropriate rules are $r_{ST}^{(fp)}$ and $r_{ST}^{(np)}$. The algorithm first executes $r_{ST}^{(fp)}$. But the NAC in the input domain blocks its application because attribute $a1$ contained

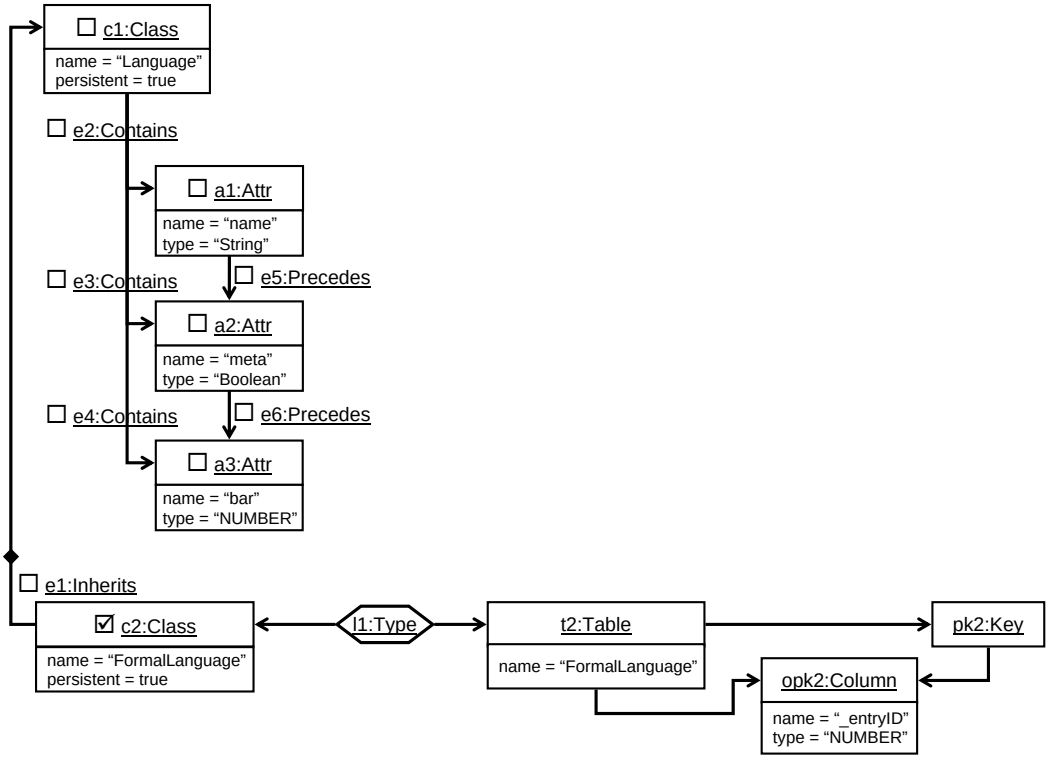


Figure 6.3.: Intermediate graph triple produced by forward translator.

in class $c1$ has already been translated. So, the algorithm proceeds and successfully executes $r_{ST}^{(np)}$ which translates $a2$, $e3$, and $e5$.

Next, the algorithm translates attribute $a3$. Both candidate rules $r_{ST}^{(fp)}$ and $r_{ST}^{(np)}$ are also appropriate. But only $r_{ST}^{(np)}$ can be executed successfully because the NAC of $r_{ST}^{(fp)}$ blocks its application. So, $a3$ is translated by $r_{ST}^{(np)}$ and $a3$, $e4$, and $e6$ are added to the set of translated elements.

The last object in the list of to-be-translated objects is $a1$. But as $a1$ has already been translated during a recursive call to procedure *translate* the algorithm instantly returns and proceeds translating the list of links $(e2, e6, e3, e5, e4, e1)$. All links except $e1$ are already translated. So, the algorithm only has to translate inheritance link $e1$. The candidate rule that finds a core match in the context of inheritance link $e1$ is $r_{ST}^{(i)}$. The context nodes $c1$ and $c2$ are already translated, so rule $r_{ST}^{(i)}$ is also appropriate. Its execution succeeds and produces a foreign key in the target domain that represents an inheritance relationship. Link $e1$ is added to the set of translated elements.

Finally, all elements have been translated by the algorithm and the produced graph triple looks like depicted in Fig. 6.4. The sequence of translation rules determined by the algorithm is $(r_{ST}^{(t)}@\emptyset, r_{ST}^{(t)}@\emptyset, r_{ST}^{(fp)}@c1, r_{ST}^{(np)}@c1, r_{ST}^{(np)}@c1, r_{ST}^{(i)}@c1c2)$.

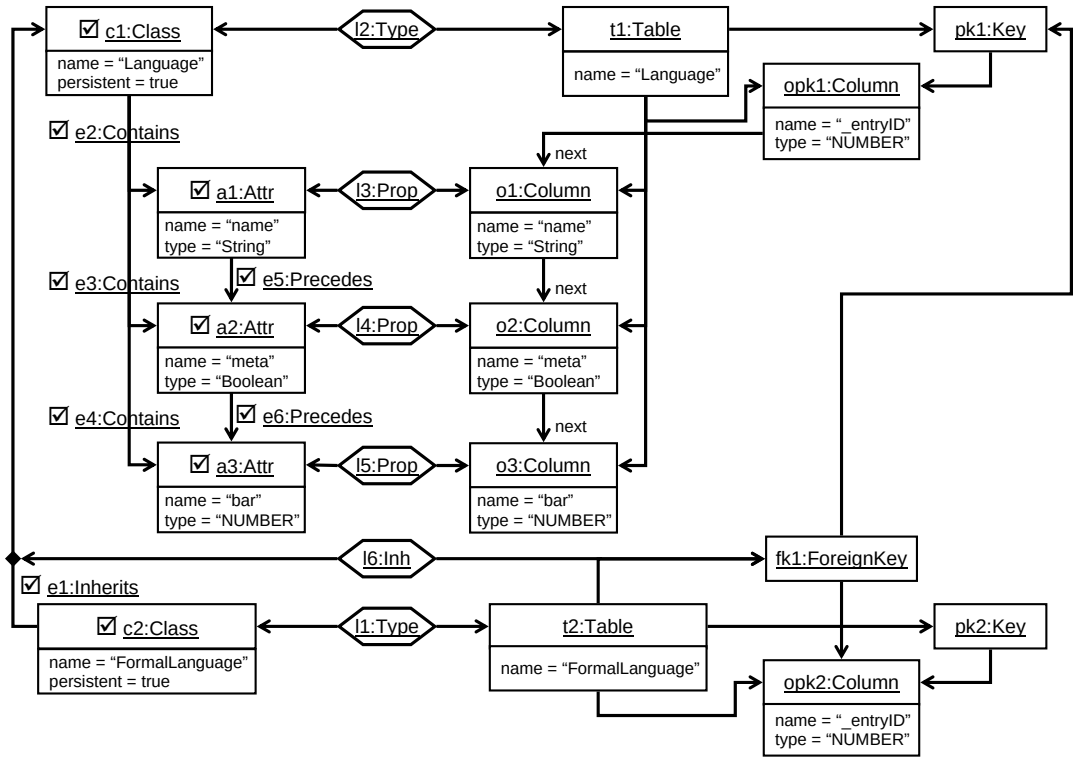


Figure 6.4.: Graph triple produced by forward translator.

6.5. Discussion of Simple Algorithm

The presented algorithm from [SK08] automatically determines a proper ordering of forward/backward rule applications even in the presence of NACs using an eager, demand-driven rule ordering approach. Therefore, it does not rely on a predefined order of the elements of an input graph or TGG productions. The algorithm uses recursive calls of the procedure *translate(Node)* to reorder node visits on demand. It is the job of the recursively defined procedure *translate(Node)* to compute the appropriate order in which rules are applied to their primary node matches in the input graph. Whenever a node of the input graph shall be translated into a new subgraph of the output graph, the procedure in a first step guarantees the following property: all context nodes in the input graph that are potentially needed by any rule that may translate the input node are determined by a core rule match and translated recursively beforehand (if possible). The algorithm translates a given input graph in a single pass into an output graph without any book keeping overhead rather efficiently.

The main drawback of the algorithm lies in the fact that we cannot guarantee its completeness for arbitrary TGGs. The algorithm does not implement complete FGT/BGT functions in the general case for the following reasons: procedure *translate(Node)* executes an arbitrarily selected rule with an arbitrarily selected match if more than one rule with more than one match can be used to translate a regarded primary node of the input graph. Furthermore, procedure *translate(Node)* uses an eager approach that translates context nodes of a just regarded input node as early as possible. As a consequence it may happen that the check

of a NAC fails due to the fact that some elements in the input graph have been translated too early. Thus we either have to modify the algorithm such that it is able to explore all derivation alternatives, which would have significant impact on its efficiency, or to use the class of TGGs with integrity-preserving productions with NACs for which we can guarantee the completeness of the developed algorithm.

The algorithm does not explicitly handle secondary elements. But in the case of TGG_{CDDS} this is necessary when translating backward. If a secondary node is passed to the algorithm this node cannot be translated. Only if the associated primary node is passed to the algorithm the secondary node is translated as a side-effect. This results in problems if a secondary node is translated due to a recursive context translation but the associated primary node is not used as context because the algorithm demands that all context elements must be translated.

In general TGG-based translators have to face the following challenges

- Simulate the simultaneous evolution of the whole graph triple.
- Parsing of input graph: which rule to apply to which node?
- Find a sequence of rules that simulates the evolution of the input graph.
- Handling of Multiple Applicable Rules.
- Handling of Multiple Matches of a rule at a given context.
- How to distinguish primary from secondary elements?
- How to determine a right translation sequence of nodes/edges that exactly mimics the TGG production application sequence?
- If context is translated recursively: how to break cycles in recursive context translation?

6.6. Advanced Graph Translation Algorithm

The algorithm depicted in Listing 4 is a modified version of the algorithm presented in [KLKS10] and is based on the algorithm in [SK08], i.e., Listing 3. This modified version additionally supports secondary elements. Likewise to the algorithm in [SK08], the algorithm from [KLKS10] uses an eager, demand-driven rule ordering approach (cf. Sect. 6.3). That is, it recursively translates all context nodes that are required by the node that is currently translated. Contrary to the algorithm in [SK08], the algorithm in Listing 4 regards multiple matches of rules in the input graph. Moreover, the algorithm is complete, it implements the dangling edge condition, and has a more sophisticated error detection.

There are other possibilities to realize a TGG translation algorithm that is able to cope with non-deterministic behavior of TGG productions. First, an algorithm could use backtracking and try other translation sequences until it finds a translation sequence that produces a correct output graph triple. But, backtracking is very expensive, i.e., not efficient in our terms, and

not necessary in general. Second, one could use *critical pair analysis*[2] to check whether the set of TGG productions is *confluent*[3]. That is, the application order of multiple applicable translation rules does not matter because the result of their application in different order is equal or isomorphic. But, it is questionable whether TGGs fulfill the confluence property in general. Our approach relies on the *local completeness criterion* introduced in Sect. 5.2.4. This criterion guarantees that the input graph, i.e., source graph or target graph, can be parsed deterministically if more than one rule is applicable because all rules translate the same elements and every match of a translation rule in the input graph is extendable to a match in the output graph.

The algorithm in Listing 4 is based on the following assumptions:

- Given a node, the algorithm is able to determine whether the node is secondary. Given a secondary node, the algorithm is able to navigate deterministically from a secondary node to its primary node.
- If a primary node is matched by a translation rule, then all to-be-created elements are uniquely determined, i.e., all additional secondary elements are matched deterministically.
- TGG production components are always *integrity-preserving*. That is, they never produce graphs that are invalid according to their well-foundness rules of the graph schema, i.e., all graphs produced by a translation rule are syntactically and semantically correct (cf. Sect. 5.2.2).

The algorithm in Listing 4 can be divided into several activities: The algorithm ensures that elements are only translated once (line 2). The algorithm detects cycles in the recursive context translation (lines 3 to 4). It handles secondary elements (lines 8 to 11). It determines appropriate rules (lines 17 to 27), then applicable rules (lines 29 to 35). It examines the determined applicable rules, whether they satisfy certain conditions (lines 37 to 44). Finally, it executes a translation rule that translates certain elements of the input graph (lines 46 to 48). In the following we will discuss these activities of the algorithm in more detail.

Cycles in Recursive Context Translation The algorithm in Listing 4 aborts with a *cycle in recursive context translation* error if an element that is currently translated is translated again due to a recursive translation call. This situation occurs, e.g., if two certain TGG production fragments are used in conjunction. Figure 6.5 depicts a TGG that contains TGG production fragments that raise such a forbidden situation.

In order to give an example, where a cycle error occurs, we pass the model depicted in Fig. 6.5 as input graph to a forward translator derived from the depicted TGG. The translator starts translating node $a1$. It marks this node as just regarded element. Afterwards, it determines the set of candidate rules. Rules 1 and 2 are candidates that also find matches

[2] A critical pair (of translation rules) is a pair of parallel dependent direct transformations [EEPT06]. According to [HEGO10], critical pairs define conflicts of rule applications in a minimal context.

[3] In a confluent transformation system, the order in which productions are applied yields the same result up to isomorphism.

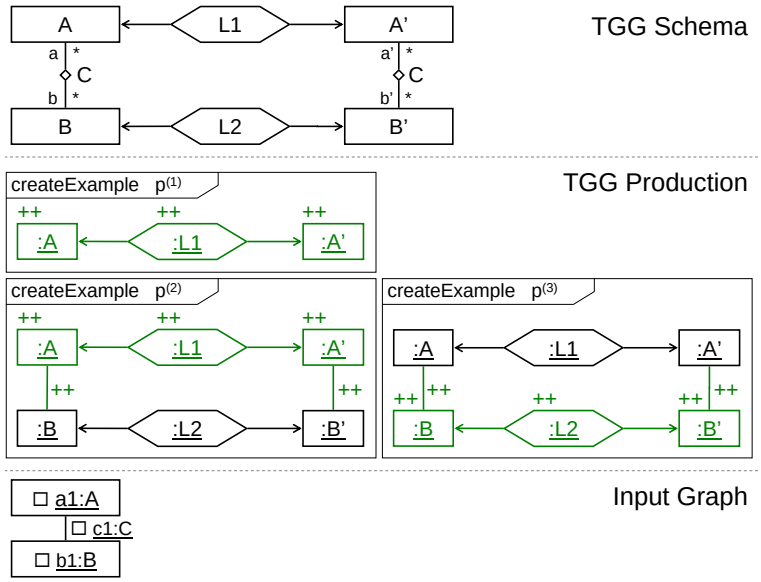


Figure 6.5.: A TGG that raises cycle errors.

in the input graph. Let us assume rule 2 is checked first. The dangling edge condition is satisfied and the additional context $b1$ needs to be translated beforehand. Therefore, $b1$ is recursively translated. The only candidate rule for translating $b1$ is rule 3. It finds a match and satisfies the dangling edge condition. In addition, it requires the context $a1$ to be translated beforehand. Therefore, $a1$ is recursively translated. The algorithm notices that $a1$ is a just regarded element and, therefore, throws a *cycle in recursive context translation* error, as it is already translated somewhere up in the translation chain.

Consequently, we forbid the combination of the patterns depicted in TGG production components $p_S^{(2)}$ and $p_S^{(3)}$ (cf. Fig. 6.5) to avoid cycle errors. Developing a statical analysis mechanism that checks a given set of TGG productions for such a combination of patterns is future work.

Translation of Secondary Elements Second, the algorithm in Listing 4 determines whether the given node is secondary utilizing operation "isSecondaryNode(Node)". If this is the case, the corresponding primary node is fetched with operation "getPrimaryNode(Node)" and actively translated, which in turn also translates the secondary node. The operations "isSecondaryNode(Node):boolean" and "getPrimaryNode(Node):Node" are derived from core rules as described in Sect. 6.2. These operations must always uniquely determine whether a given node is primary or secondary and uniquely determine the corresponding primary node of a given secondary node respectively.

If a node is detected to be secondary then the translation of its primary node always translates the secondary node, too. Note that the corresponding primary node of a secondary node has to be actively translated. Otherwise the algorithm would fail if the secondary node is used as context in another rule, i.e., is translated during a call to a recursive context

translation, because such a call must always translate every element that is required as context. A "lazy" translation is not sufficient in this case.

An implicit property of secondary elements is that no rule exists that is able to translate the secondary element passing it as primary node to a candidate rule. The specificator of a set of TGG productions has to ensure this property. Developing a statical analysis mechanism that checks a given set of TGG productions for this property is up to future work. Note that the algorithm in Listing 4 never translates a node actively if it detects that this node is a secondary node. Instead, it determines and translates the corresponding primary node which implicitly translates the secondary node and further secondary elements. Consequently, the translation operation of the secondary node returns (cf. line 11 of Listing 4) right after translating the primary node.

Determination of Appropriate Rules Third, the algorithm determines *appropriate rules* from the set of *candidate rules*. A rule is a candidate if its primary node in the input domain is type compatible with the to-be-translated primary node. An appropriate rule has at least one *core match* which contains the primary node. A core match satisfies Def. 24 (2), i.e., all to-be-translated elements are not translated yet. If multiple matches of one rule in the input graph exist[4], the algorithm checks for every match if the rule is appropriate. In order to be appropriate, every context node required by the primary node is recursively translated. But, the context is only translated if the dangling edge condition would be satisfied afterwards (cf. Def. 24 DEC(1)). If the DEC fails this indicates that the current node has incident edges that are not producible by the TGG if the current rule is applied. Therefore, the algorithm skips the current core match and proceeds with the next core match or rule candidate.

Note that the algorithm in Listing 4 is greedy in the sense that it translates every element used as context of every match of every candidate rule. This might translate more elements then required as context elements by the applicable rule that is finally applied at an appropriate match including the to-be-translated primary node.

Determination of Applicable Rules Fourth, the algorithm determines *applicable rules* from the set of appropriate rules. The application condition Def. 24 DEC(1) has to be reassured because it might have been invalidated due to potential competing recursive context translations. In addition, the core match of an applicable rule must be completed in the rule's left-hand side (i.e., input, link, and output domain) and the NACs in the output domain must not block.

Examination of Applicable Rules If no applicable rule is determined then either a match exists in the input domain but it may not be completed or the to-be-translated node is not even locally translatable. In the first case the set of TGG productions violates the local completeness criterion (cf. Def. 21) and a corresponding error is thrown. This error states that the TGG specification is erroneous. In the latter case the input graph is invalid and a corresponding exception is thrown. This exception states that the input graph given to the translation algorithm is invalid and, therefore, cannot be translated successfully.

[4] In TGG_{CDDS} at most one core match exists for any rule in a valid input graph.

```
1  procedure void translate(n: Node) {
2    if (n ∈ translatedElements) return;
3    else if (n ∈ justRegardedElements)
4      throw CycleInRecursiveContextTranslationError(n, justRegardedElements);
5    else {
6      justRegardedElements.add(n);
7
8      if (isSecondaryNode(n)) {
9        translate(getPrimaryNode(n));
10       justRegardedElements.remove(n);
11       return;
12     }
13
14     nodeLocallyTranslatable: boolean = false;
15     appropriateRules, applicableRules: PairSet<Rule, Match> = ∅;
16
17     candidateRules: RuleSet = select rules where r.primaryInputNode.type equals n.type;
18     forall (rule ∈ candidateRules) { // collect appropriate rules and core matches
19       compute core matches of rule in inputGraph with n as primary node;
20       forall (cm ∈ core matches) { // Def.24(2): m'_I(R_I\L_I) ∩ TX = ∅ satisfied!
21         if (not isDECSatisfied(n, join(cm.toBe, cm.context))) // m'_I(R_I\L_I) ∪ m'_I(L_I)
22           { continue; } // do not translate context if Def.24(DEC(1)) would be violated
23         forall (contextNode ∈ context elements of core match)
24           { translate(contextNode); } // recursively translate required context
25         if (all context elements of core match are translated)
26           { appropriateRules.add(rule, cm); } // Def.24(1): m'_I(L_I) ⊆ TX satisfied!
27     } } // end of appropriate rule at core match with n as primary node calculation
28
29     forall ((rule, cm) ∈ appropriateRules) { // collect rules applicable at full match
30       // reassure Def.24(DEC(1)): may be violated due to competing context translation
31       if (not isDECSatisfied(n, cm.toBe)) { continue; }
32       nodeLocallyTranslatable = true;  // now n must be translatable due to Def.21
33       if (cm can be completed in other domains and NACs in output domain don't block)
34         { applicableRules.add(rule, full match); }
35     } // end of applicable rule at full match with n as primary node calculation
36
37     if (applicableRules.isEmpty()) { // node is not translatable
38       if (nodeLocallyTranslatable) // match could not be completed in other domain(s)
39         throw LocalCompletenessCriterionError(n, appropriateRules); // Def.21 violated!
40       else
41         throw InputGraphNotPartOfDerivableGraphTripleException(n); // G_input ∉ L(GG_I)
42     }
43     if (not applicableRules.matches->forAll(m1, m2 | m1 <> m2 implies
44       m1.cm.toBe = m2.cm.toBe)) throw CompetingCoreMatchesError(n, applicableRules);
45
46     select one rule/match pair from applicableRules;
47     apply rule at match;  // evolve GT ⇝ GT' with r_IO@m_IO
48     translatedElements.add(elements of inputGraph translated by rule); // m'_I(R_I\L_I)
49     justRegardedElements.remove(n);
50 } }
51
52 procedure boolean isDECSatisfied(node: Node, toBeTranslated: ElementSet) {
53   translatedElements' = translatedElements ∪ toBeTranslated; // TX' (cf. Def.24(3))
54   select all incident edges e of node where (e ∉ translatedElements')
55                                       and (s(e) or t(e) ∈ translatedElements')
56   forall (e ∈ selected incident edges)
57     { if (not (DEC(1) satisfied for e)) { return false; } } // ensure Def.24(DEC(1))
58   return true; // all edges translatable
59 }
```

Listing 4: Algorithm that handles NACs and checks for dangling edges.

If multiple rules are applicable at some completed match, the algorithm ensures that their to-be-translated elements in the core match are identical. Otherwise it aborts with an error indicating that there are competing core matches, i.e., there are different to-be-translated elements, as this might lead into dead-ends (cf. Sect. 5.3.2). It is up to the developer of a set of TGG productions to guarantee that this will never happen in practice. Consequently, the error reports that the TGG specification is erroneous.

Execution of Translation Rules Finally, the algorithm translates the primary node. It selects one entry from the set of applicable rules, applies the rule at its match, and extends the set of translated elements (cf. Def. 24 (3)).

Note that the translation of a primary node always translates its secondary elements as well. Moreover, if the translation operation was invoked by a call to "translate(Node)" in line 9 of the algorithm in Listing 4 then the secondary node that was used to determine the primary node is translated if the translation of the primary node succeeds. This is guaranteed due to the property that must hold for operation "getPrimaryNode(Node):Node" (i.e., unique determination of primary node) and the construction of translation rules. Translation rules are based on core rules, likewise to the construction of operation "getPrimaryNode(Node):Node" which is also based on core rules. Consequently, if operation "getPrimaryNode(Node):Node" was able to uniquely determine the primary node from a given secondary node, a distinguished translation rule must exist that translates the primary node as well as the original secondary node. Thanks to the examination of applicable rules, as described in the preceding paragraph, the algorithm would throw a *competing core matches* error if there exists any other applicable rule that would translate other elements than the distinguished rule. So, the distinguished rule or any other equivalent rule that translates the same elements is executed in order to translate a primary node corresponding to a secondary node given by a call to line 9. Consequently, we have shown that a secondary node is translated due to the call in line 9 if the translation of its primary node succeeds, i.e., no exception or error is thrown.

Implementation of Dangling Edge Condition The implementation of the dangling edge condition in procedure *isDECSatisfied(Node, ElementSet)* is straightforward. The first parameter is the node to be checked for dangling edges. The second parameter is a set of elements that should be added temporarily to the set of current translated elements. All edges that are incident to the given node and are not contained in the temporary set of translated elements TX' but source or target node of this edge are in TX' are checked for the dangling edge condition (cf. Def. 24 (DEC(1))). If at least one of these edges violates the dangling edge condition then the procedure returns immediately stating that the dangling edge condition is not satisfied.

6.7. Backward Translation Example

Now, we will discuss a translation process with the backward translator derived from TGG_{CDDS} which uses the advanced algorithm in Listing 4. The input graph given to the translator is

the database schema depicted in Fig. 6.6. This database schema is the result of the forward translation discussed in Sect. 6.4.

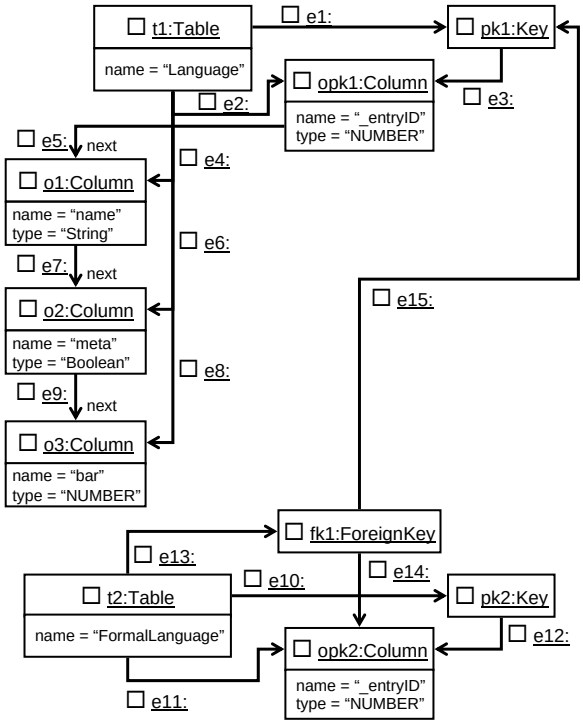


Figure 6.6.: Input graph given to backward translator.

Let us assume the translation order of object nodes is $(opk1, fk1, pk1, t1, pk2, opk2, t2, o3, o1, o2)$. Then, the algorithm starts the translation process with column $opk1$, which is the column used to store primary keys of table $t1$. The algorithm first checks whether this node is a secondary node. Therefore, it utilizes the backward core rules (cf. Fig. 6.1). It selects those core rules where the type of a secondary node equals the type of $opk1$, i.e., *Column*, and where a match of this rule is found in the input graph. According to TGG_{CDDS} nodes of type *Column* or *Key* have the potential for being secondary nodes. In the case of $opk1$ core rule $cr_{TS}^{(t)}$ finds a match using $opk1$ as secondary node. Therefore, the algorithm identifies $opk1$ as secondary node and translates its according primary node, table $t1$. BGT rule (t) (i.e., $r_{TS}^{(t)}$) is the only candidate that has the capability to translate $t1$. The algorithm checks whether application of rule (t) satisfies DEC(1). Therefore, it determines the elements in G_{input} created by the corresponding local rule $r_T^{(t)}$ (i.e., $\{t1, opk1, pk1, e1, e2, e3\}$) and joins these elements, the required context elements (i.e., $\emptyset$), and the set of currently translated elements (i.e., $\emptyset$) which results in set $TX' := \{t1, opk1, pk1, e1, e2, e3\}$. Then, it checks whether condition DEC(1) is satisfied for all not translated incident links of node $t1$, i.e., $inc(t1) := \{e4, e6, e8\}$. According to Sect. 5.3.3, the required tuple for $inc(t1)$ is $(Table, Contains, Column, e1)$. So, DEC(1) is satisfied because $LOCC_T(1)$ contains this tuple (cf. Table 5.1 in Sect. 5.3.3). Since $t1$ does not have any context, no additional elements need to be translated and rule (t) is marked *appropriate*. Moreover, its core match can be completed to a full match. Therefore, $r_{TS}^{(t)}$ is an

applicable rule. As it is the only applicable rule it is applied at the complete match which translates table $t1$, key $pk1$, column $opk1$, links $e1$, $e2$, and $e3$ and creates a corresponding class in the source domain.

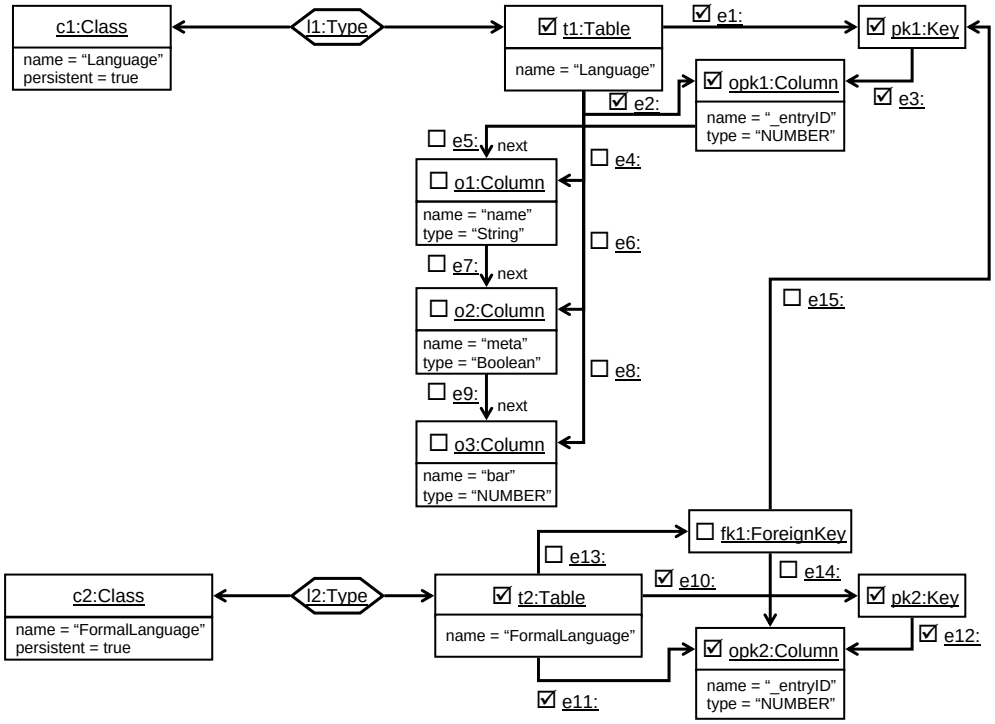


Figure 6.7.: Intermediate graph triple produced by backward translator.

The algorithm proceeds translating by selecting foreign key $fk1$ from the set of remaining object nodes. Candidate rules are BGT rules (r) and (i) but only (i) finds a match in the input graph. The DEC check succeeds because no incident untranslated links remain if $fk1$ would be translated. Now, all context nodes of $fk1$, i.e., $\{t1, pk1, opk1, e1, e3, t2, pk2, opk2, e10, e12\}$, are recursively translated. As $t1$, $pk1$, $opk1$, $e1$, and $e3$ are already translated these nodes are skipped and the algorithm proceeds translating table $t2$. Likewise to table $t1$, table $t2$ is translated with BGT rule (t) into a corresponding class as DEC(1) is satisfied due to the existence of $LOCC_T$ entry $(Table, RelatesToForeignEntriesVia, ForeignKey, e1)$, i.e., link $e13$ is translatable afterwards. So, $t2$ is translated by $r_{TS}^{(t)}$ and $t2$, $pk2$, $opk2$, $e10$, and $e12$ are added to the set of translated elements. The graph triple now looks like depicted in Fig. 6.7.

After all context elements of $fk1$ are translated the algorithm resumes translating $fk1$ with the appropriate BGT rule (i), whose match can be completed in the other domains. Moreover, the NAC in the output domain does not block. So, after translating $fk1$ with BGT rule (i) the elements $fk1$, $e13$, $e14$, and $e15$ are added to the set of translated elements.

Now, the algorithm proceeds translating the set of remaining object nodes. Nodes $pk1$, $t1$, $pk2$, $opk2$, and $t2$ are already translated so the algorithm proceeds translating column $o3$. This column is not a secondary node, so the algorithm tries to translate it as primary node. BGT rules (fp) and (np) are candidate rules for translating $o3$. The same core match, which

requires table $t1$ and column $o2$ as context, exists for both rules. DEC(1) is satisfied as there are no untranslated incident edges after $o3$ would be translated.

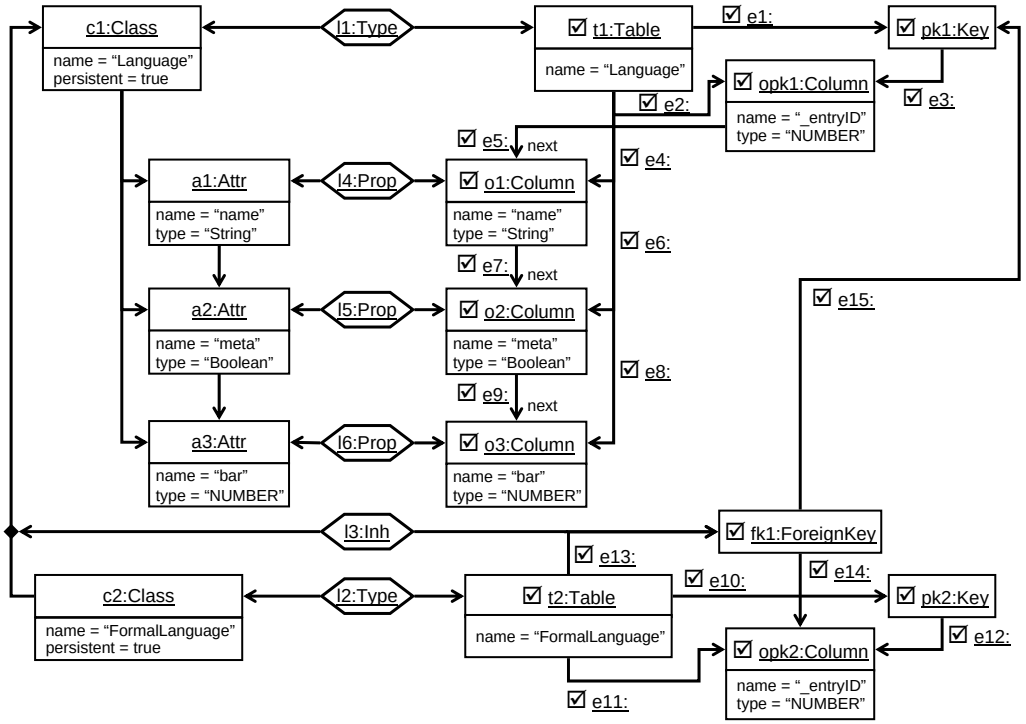


Figure 6.8.: Graph triple produced by backward translator.

But before translating $o3$, first its context $o2$ has to be translated. Another recursive context translation step has to be executed when $o2$ is translated because it requires $o1$ as already translated context element. So, the algorithm reorders the elements to be translated such that first it translates $o1$, then $o2$, then $o3$. In all three cases the candidate rules are BGT rules (fp) and (np). DEC(1) is satisfied for $inc(o1) = \{e7\}$ and $inc(o2) = \{e9\}$ as $(Column, Precedes, Column, e1) \in LOCC_T(1)$. When translating $o1$ the match of rule (fp) is the only one that can be completed in the source domain, i.e., the only applicable rule. Rule (np) is not applicable as $o1$ is the first attribute which is added to the class that corresponds to table $t1$. Hence, rule (np) is the only rule that can be completed in the source domain when translating $o2$ and $o3$ as the NAC in the source domain blocks the application because an attribute is already present in the class corresponding to table $t1$. Consequently, both locally-applicable rules $r_{TS}^{(fp)}$ and $r_{TS}^{(np)}$ are *disjoint applicable* in this case (cf. discussion in Sect. 5.3.2). After translating $o1$, $o2$, and $o3$ the graph triple looks as depicted in Fig. 6.8.

Finally, the algorithm translates all links belonging to the input graph. But, all links have already been translated earlier while translating the object nodes. Therefore, the algorithm simply skips the translation of the link nodes $e1$ to $e15$.

So, the algorithm has successfully translated all objects and links of the input graph to a corresponding output graph. The sequence of applied BGT rules
$SEQ(r_{TS}) = (r_{TS}^{(t)}@\emptyset, r_{TS}^{(t)}@\emptyset, r_{TS}^{(i)}@t1t2, r_{TS}^{(fp)}@t1, r_{TS}^{(np)}@t1, r_{TS}^{(np)}@t1)$ constructed in this exam-

ple translates the primary nodes in this order: $(t1, t2, fk1, o1, o2, o3)$. In conjunction with the also constructed correspondence graph a graph triple GT_{out} was produced which is equivalent to the graph triple produced by the forward translator (cf. Fig. 6.4).

6.8. Properties of Advanced Translation Algorithm

The next theorems state that translators based on the advanced algorithm specified in Listing 4 terminate, are efficient as well as correct and complete with respect to their TGG if the algorithm never aborts with an exception for any given valid input graph. If the algorithm aborts with an exception then $G_{input} \notin \mathcal{L}(GG_I)$ and if the algorithm aborts with an error then the TGG specification is erroneous, i.e., does not satisfy the conditions stated throughout this contribution.

6.8.1. Termination

The algorithm terminates if it stops after a finite number of steps and returns a result or throws an error or exception. First, we argue that the procedures of the algorithm framework *evolve(GraphTriple)* and *translate(GraphTriple)* terminate. Then, we show that the procedures of the algorithm *translate(Node)* and *isDECSatisfied(Node, ElementSet)* terminate. In general, a procedure terminates if every call to a subroutine terminates and if loops are somehow bounded, e.g., if they iterate through a finite number of elements.

Theorem 2. *Termination of Graph Translation.*
The translation algorithm in Listing 4, which is composed of procedures "evolve(GraphTriple)", "translate(GraphTriple)", "translate(Node)", and "isDECSatisfied(Node, ElementSet)", terminates.

Proof.
Procedure *evolve* does not have any loops or recursive calls. So, it terminates if all subroutine calls terminate. This has to be proven for *translate(GraphTriple)* but is true for all other subroutines.

Procedure *translate(GraphTriple)* has two loops that iterate through a finite number of objects and links of a given input graph and calls procedure *translate(Node)* for every such object and link. Therefore, this procedure terminates if the calls to *translate(Node)* terminate.

Procedure *translate(Node)* in Listing 4 terminates. This is due to the fact that the set of TGG productions is finite and so are the set of translation rules r_{IO} and the set of core rules. Moreover, the number of elements in a graph is finite, and every element is only translated once. Therefore, all recursive calls to *translate(Node)* terminate if the given node is already translated or if it is successfully translated or the translation produces an error or exception. Recursion cycles are detected and explicitly broken by the algorithm, i.e., the algorithm cannot run into infinite recursion calls. The number of matches determined for any rule is finite, as well as the number of context elements of a core match. Therefore, the loops of procedure *translate(Node)* are bounded and terminate.

Procedure *isDECSatisfied(Node, ElementSet)* terminates because the number of incident edges of a node is finite. □

After the translation algorithm terminates it may be in three different modes. The first mode is the desired mode for the translation of a valid input graph: the translation was successful and the output graph triple is part of the language defined by the TGG. In the second and third mode we distinguish between *exception* and *error*. An exception is less severe and states that the user has made a fault, e.g., by passing an input to the translator that is not translatable. In this case, the algorithm throws an *input graph not part of derivable graph triple* exception (cf. line 15 in Listing 1 and line 41 in Listing 4). An error is more severe stating that the underlying TGG specification is erroneous and it has to be reviewed by a TGG designer. The translation algorithm has the following error states:

- *cycle in recursive context translation* error (cf. line 4 in Listing 4): A recursive call to procedure "translate(Node)" has produced a cycle, i.e., an element that is currently in the translation chain is about to be translated again. This error is a special case as it might occur due to an invalid TGG specification (cf. discussion on cycle errors in Sect. 6.6), but also due to an invalid input graph. A situation where an invalid input graph raises a *cycle in recursive context translation* error is depicted in Fig. 6.9. The TGG schema is already known from the TGG_{CDDS} example. The two TGG productions satisfy the condidtions that must hold for TGG productions. The first TGG production is a simplified version of TGG production "createType" from TGG_{CDDS} (cf. Fig. 4.6 in Sect. 4.4). The second TGG production creates a subclass and links this subclass to the table that corresponds to the superclass[5].

  An invalid input graph consisting of two classes $c1$ and $c2$ with an inheritance cycle (made up by inheritance links $i1$ and $i2$) is given to a forward translator that utilizes the algorithm in Listing 4. This input graph is not producible by the two TGG productions "createType" $p^{(1)}$ and "createSubtype" $p^{(2)}$, i.e., it is invalid according to the TGG specification. The translator starts translating node $c1$. It marks this node as just regarded element. Afterwards, it determines the set of candidate rules. Rules 1 and 2 are candidates that also find matches in the input graph. Let us assume rule 2 is checked first. The dangling edge condition is satisfied and the additional context $c2$ needs to be translated beforehand. Therefore, $c2$ is recursively translated. The two candidate rules for translating $c2$ are rules 1 and 2. Rule 1 does not satisfy the dangling edge condition, but rule 2 does. Rule 2 finds a match and in addition, it requires the context $c1$ to be translated beforehand. Therefore, $c1$ is recursively translated. The algorithm notices that $c1$ is a just regarded element and, therefore, throws a *cycle in recursive context translation* error, as it is already translated somewhere up in the translation chain. As already mentioned, the error does not state a TGG specification error in this case but a user error. The only way to prevent such an error is to add constraints to the metamodel of the TGG specification that forbids cycles in input graphs and, therefore, allows to detect invalid input graphs in the constraint check in line 6 of Listing 1. A constraint

[5] This TGG production is a well-known example for a different realization of inheritance of classes (and tables) in the CDDS example (cf. [Kön05, EEE+07, SK08, KLKS10]).

check could then be added to the algorithm in Listing 4 that checks for validity of the input graph before throwing a cycle error.

- *local completeness criterion* error (cf. line 39 in Listing 4): The TGG specification does not satisfy the local completeness criterion as given in Defs. 21 and 22 in Sect. 5.2.4.
- *competing core matches* error (cf. line 44 in Listing 4): Two or more translation rules compete translating the same primary node but with different secondary elements, i.e., incident edges or adjacent nodes.
- *TGG contains integrity-destroying productions* error (cf. line 14 in Listing 1): The TGG is not composed of integrity-preserving productions only (as required by Def. 17 in Sect. 5.2.2) but contains productions that destroy the integrity of a graph.

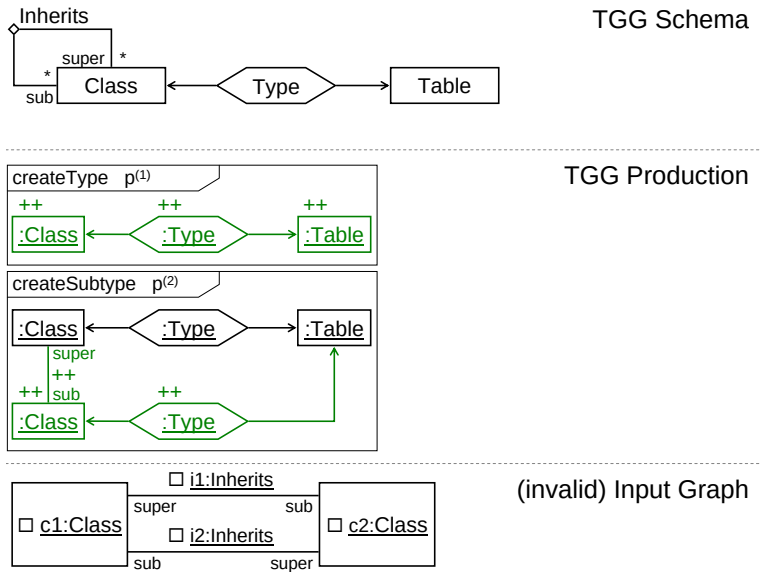


Figure 6.9.: A TGG that raises a cycle error due to an an invalid input graph.

In the case of an error that is reported by the algorithm, the TGG is assumed to violate one of the properties it has to satisfy. Therefore, the TGG specification has to be reviewed and the operational rules have to be derived again in order to be applicable successfully by the algorithm.

6.8.2. Efficiency

In the following theorem we will discuss the efficiency of the algorithm in Listing 4. We will see that it has a polynomial space and time complexity in the worst case. This is due to the fact that the algorithm processes the elements of the input graph in a given order such that no element is regarded more than a constant number of times and translated only once. Multiple translation alternatives are not explored because this indicates a TGG specification error.

Theorem 3. *Efficiency of Graph Translation.*
The algorithm in Listing 4 has worst case runtime complexity of $O(n^k)$ *with* n *being the number of nodes of* G_{input} *and* k *being a constant that depends on the regarded TGG.*

Proof.
Sketch:

(1) The algorithm just loops through the set of all n nodes of the input graph; the implicit reordering of the translation of input graph elements in the loop for not yet translated context elements of a just regarded graph element does not affect its runtime complexity.

(2) The book keeping overhead of the algorithm is neglectible and the execution time for basic graph operations like traversing an edge or creating a new graph element is bounded by a constant (otherwise we should add a logarithmic or linear term depending on the implementation of the underlying graph data structure).

(3) The worst case execution time of all needed rules applied to a given (primary) input graph node is $(n + n')^{k-1}$, where n' is the number of nodes of the output graph, and k is the maximum number of elements of any applicable rule. In the worst case the match of the primary node is extended by testing all possible $(n + n')^{k-1}$ permutations of source/target graph elements.

(4) Furthermore, $n' \leq c * n$ for a given constant c that is the maximum number of new nodes of the output component of any TGG production.

(5) The recursive calls to procedure "translate(Node)" in lines 9 and 24 of Listing 4 do only translate the affected nodes of the input graph once. The procedure returns immediately if the node that is to be translated is already translated (cf. line 2 of Listing 4). If the translation fails then the algorithm aborts with an error or exception. Otherwise the translation is successful and the affected nodes are added to the set of translated elements and are never translated again.

□

In the case of TGG_{CDDS}, the complexity of a forward translation is $O(n^2)$ for the following reasons: The worst case execution time of its rules (cf. Fig. 5.6) is $O(n') \leq O(n)$, with n' being the maximum number of columns of a table in the output graph, due to the fact that the primary node is known and rules (t) and (i) have a constant execution time, whereas rules (fp), (np), and (r) have to determine the last column node of a table node. Assuming that all nodes of the output graph are columns of the just regarded table $n' \leq n$ nodes have to be inspected in the worst case.

6.8.3. Correctness

The following theorem states that the algorithm is correct with respect to a TGG specification. The *correctness* of a forward or backward translator is guaranteed if it translates an input graph into a graph triple that is always an element of the language $\mathcal{L}(TGG)$ defined by the

TGG. Furthermore, the output graph is an element of the language $\mathcal{L}(TG_O, \mathcal{C}_O)$ defined by the output type graph and the set of constraints.

Theorem 4. *Correctness of Graph Translation.*
Let $G_I \in \mathcal{L}(TG_I, \mathcal{C}_I)$ be an input graph (either G_S or G_T)
and G_O be an output graph (either G_T or G_S).
If $FGT(G_S \leftarrow G_\emptyset \rightarrow G_\emptyset) = (G_S \leftarrow G_C \rightarrow G_T)$
is a not aborting complete translation of G_I with the algorithm in Listing 4 then:
(1) $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ and
(2) $G_O \in \mathcal{L}(TG_O, \mathcal{C}_O)$.
If $BGT(G_\emptyset \leftarrow G_\emptyset \rightarrow G_T) = (G_S \leftarrow G_C \rightarrow G_T)$
is a not aborting complete translation of G_I with the algorithm in Listing 4 then:
(1) $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ and
(2) $G_O \in \mathcal{L}(TG_O, \mathcal{C}_O)$.

Proof.
Sketch:

(1) $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ is a direct consequence of Theorem 1 (cf. Sect. 5.2.2) and the fact that $G_I \in \mathcal{L}(TG_I, \mathcal{C}_I)$. As a consequence the simulated application of TGG productions without NACs in the input domain does not have any effect concerning the applicability of translation rules.

(2) The behavior of translation rules on the output side is identical with the behavior of the related TGG production: i.e., a rule finds a match on the output side iff the related TGG production has the same match.
$G_O \in \mathcal{L}(TG_O, \mathcal{C}_O)$ is then a direct consequence of (1).

□

6.8.4. Completeness

The next theorem states that the algorithm in Listing 4 is complete with respect to a TGG specification. The *completeness* property demands that for every graph triple that is an element of $\mathcal{L}(TGG)$, a translator is able to produce this graph triple (or an equivalent one) given the graph of the graph triple, which belongs to the translator's input domain.

Theorem 5. *Completeness of Graph Translation.*
Let $(G_S \leftarrow G_C \rightarrow G_T) \in \mathcal{L}(TGG)$ and let us assume that the execution of the algorithm in Listing 4 does not abort with any error. Then, we can guarantee that graphs G_C^*, G_T^* and G_S^*, G_C^* exist such that:
$FGT(G_S \leftarrow G_\emptyset \rightarrow G_\emptyset) = (G_S \leftarrow G_C^* \rightarrow G_T^*) \in \mathcal{L}(TGG)$ and
$BGT(G_\emptyset \leftarrow G_\emptyset \rightarrow G_T) = (G_S^* \leftarrow G_C^* \rightarrow G_T) \in \mathcal{L}(TGG)$ respectively;
i.e., the algorithm terminates without throwing any exception.

Proof.
(by induction) Sketch:
Let $GT_{out} \in \mathcal{L}(TGG)$ be a graph triple that has been derived using a sequence of derivation steps $SEQ_{i=1}^{n}(p_i) = ((p@m)_1, \ldots, (p@m)_n)$ of length n and let $SEQ_{i=1}^{n}(r_{I,i}) = ((r_I@m_I)_1, \ldots, (r_I@m_I)_n)$ be the projection of the regarded sequence of graph triple derivation steps on its input graph. Furthermore, let $SEQ_{i=0}^{j}(r_{IO,i})$ with $0 \leq j \leq n$ be the sequence of the first j translation rule applications $((r_{IO}@m_{IO})_1, \ldots, (r_{IO}@m_{IO})_j)$ generated by the algorithm that exactly mimics the derivation of GT_{out}.

Case 1, $j = 0$: A translation rule sequence of length 0 trivially mimics the derivation of the empty graph triple $GT_{\emptyset}$.

Case 2, $0 < j < n$: We have to show that the algorithm extends the given sequence of rule applications of length j to a sequence of length $j+1$ such that it simulates either the original sequence of TGG productions $SEQ(p)$ or a slightly modified sequence $SEQ(p^*)$ that still generates the same input graph. Let $(v_I)_i$ be the primary node of the input graph of each rule application $(r_{IO}@m_{IO})_i$ and TGG production application $(p@m)_i$ with $1 \leq i \leq j$. Let v be the next to-be-translated primary node which is selected by the algorithm[6]. Furthermore, we assume that the algorithm has already translated successfully the context nodes of all rules that might be able to translate node v.

Case 2.1, $v = (v_I)_{j+1}$: Due to the fact that the algorithm does not throw a *competing core matches* error we can safely assume that there exists at most one set of translation rules with the same to-be-translated elements in their core match including node v. Furthermore, we know that there exists at least one rule with $p^-_{I,id}$ (that is the input component of the translation rule derived from p_{j+1}) that matches node $v = (v_I)_{j+1}$. The local completeness criterion (cf. Def. 21 in Sect. 5.2.4) guarantees that the algorithm finds a TGG production application $p^*@m^*$ that corresponds to one of the translation rules r^*_{IO} that is able to handle the translation of the selected node v. Applying Def. 21 multiple times we can generate a new sequence of TGG production applications $SEQ_{i=1}^{n}(p^*_i)$ such that:
$1 \leq i \leq j$: $(p^*@m^*)_i = (p@m)_i$
$i = j+1$: $(p^*@m^*)_i = p^*@m^*$
$j+1 < i \leq n$: $(p^*@m^*)_i$ is a new production application that mimics $(p@m)_i$
As a consequence the algorithm is able to create a sequence of translation steps $SEQ(r^*_{IO})$ of length $j+1$ that has the same properties as the given sequence of translation steps $SEQ(r_{IO})$ of length j w.r.t. the new sequence $SEQ_{i=1}^{n}(p^*_i)$ that replaces $SEQ_{i=1}^{n}(p_i)$.

Case 2.2, $v \neq (v_I)_{j+1}$: Due to the fact that the selected node v is not yet translated and that $(v_I)_1, ..., (v_I)_n$ is the complete set of all primary nodes of the given input graph (generated by the given sequence of TGG production applications) there exists an index $k > j+1$ such that $v = (v_I)_k$. Let $(p^-_{I,id})_k$ be the input component of the translation rule derived from $(p@m)_k$. We know that all context nodes potentially required by $(p^-_{I,id})_k$ are already translated. Again relying on the fact that the algorithm does not throw any error and on Def. 21 we know that a rule r^*_{IO} exists, derived from a production p^*, which is able to translate the given primary

[6] If v is a secondary node then the algorithm is able to uniquely determine its primary node v' and it translates v' instead of v. The secondary node v is implicitly translated due to the translation of its primary node v' (cf. discussion at the end of Sect. 6.6)

node v. Using the same line of arguments as in case 2.1 we can construct a new sequence of TGG productions p^* of length n with the same properties as listed above. As a consequence the algorithm is again able to create a sequence of translation steps $SEQ(r^*_{IO})$ of length $j+1$ that has the same properties as the given sequence of translation steps of length j.

Case 3, $j = n$: The translation rule sequence mimics the complete derivation of the input graph, i.e., generates a valid translation into a graph triple GT^*_{out} that has the same input graph as GT_{out} but may have different correspondence and output graphs then GT_{out}. □

6.8.5. Consequences

The consequence of the proof sketches is as follows. If we are able to show for a given correct TGG that derived translators never abort with an error then:

(1) The presented algorithm terminates.

(2) The presented algorithm can be executed efficiently (polynomial complexity) as long as the matches of all translation rules can be computed efficiently.

(3) Forward and backward translation results are correct, i.e., do only produce graph triples that belong to the language of the regarded TGG.

(4) Forward and backward translations are complete, i.e., will always produce a result for a given input graph if the language of the regarded TGG contains a graph triple that has this input graph as a component.

Finally, our running example shows that in the general case the result of a graph translation is not uniquely determined up to isomorphism, i.e., sets of TGG productions needed in practice often do not satisfy any (local) confluence criteria. Therefore, it was of importance to develop an efficiently working graph translation algorithm that does not rely on (local) confluence criteria of TGG productions or translation rules, but nevertheless fulfills the initially presented expressiveness, correctness, and completeness properties, too!

6.9. Consistency Check Algorithm

The consistency of a graph triple may have been corrupted if one of the graph components is modified after the initial translation process. A consistency check algorithm detects inconsistencies that arise due to these modifications. Therefore, a consistency check algorithm can be utilized by a process that performs incremental updates, i.e., repairs inconsistencies due to modifications of graph components.

The consistency check algorithm described in this section utilizes *consistency check rules* (*CC rules* r_{CC}). Likewise to forward and backward graph translation rules, a CC rule is derived from a TGG production. A CC rule checks for a given TGG link contained in the correspondence graph if its current situation in the graph triple is consistent with the situation defined in an according TGG production. That is, starting with a given TGG link, a CC rule

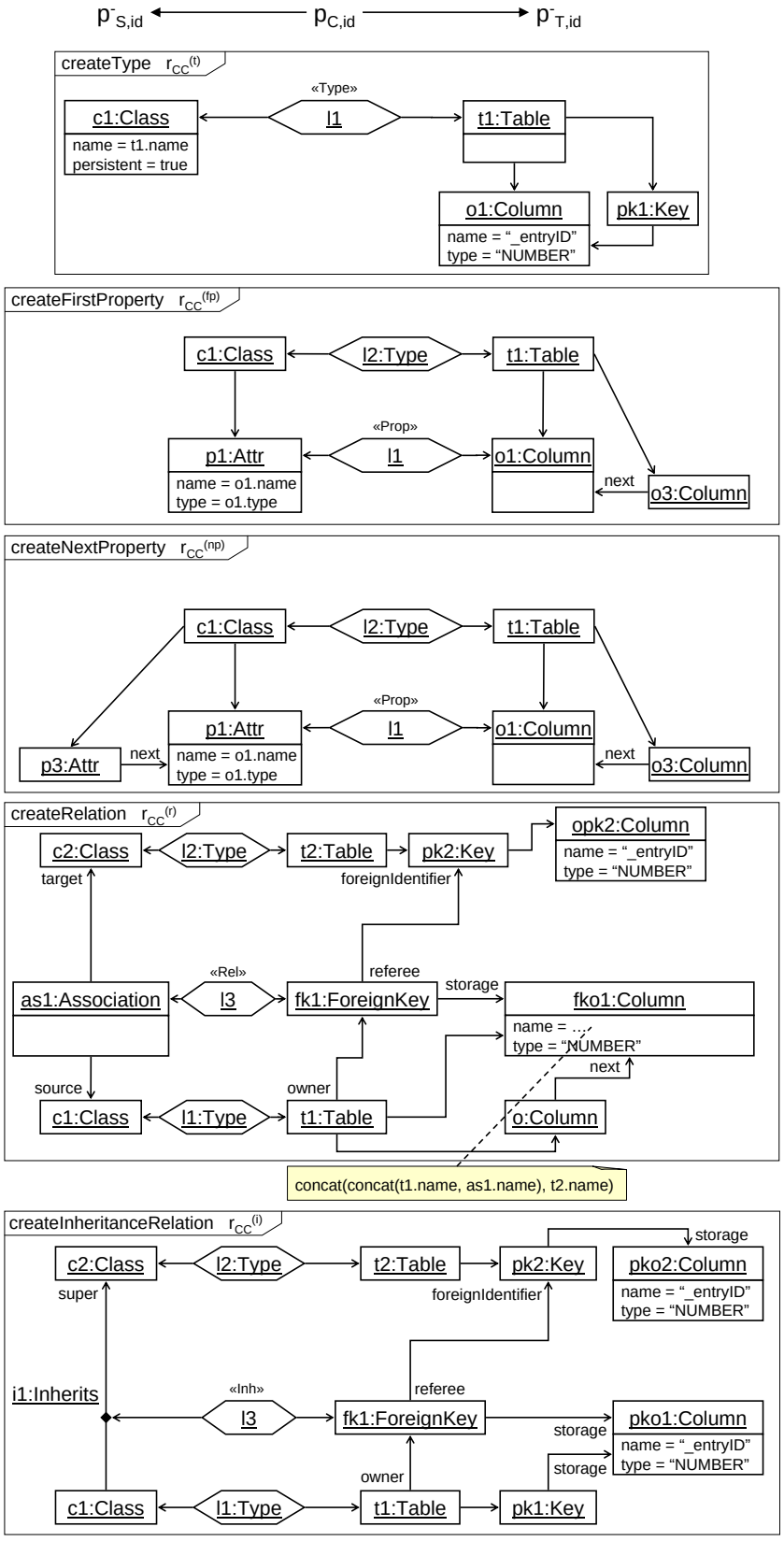


Figure 6.10.: Consistency check rules r_{CC} derived from TGG_{CDDS}.

checks whether it finds some match in the graph triple that corresponds to the simultaneous evolution process defined for this TGG link in an according TGG production.

The consistency check rules derived from TGG_{CDDS} are depicted in Fig. 6.10[7]. The components of a CC rule are the components of its TGG production where NACs have been removed. The elements that are created by the TGG production are checked for existence in a CC rule, i.e., the components of a CC rule r_{CC} are $(p^-_{S,id} \leftarrow p_{C,id} \rightarrow p^-_{T,id})$. The derivation process of CC rules from a TGG production is quite similar to the derivation process of FGT rules and BGT rules (cf. Sect. 5.2.3). The derivation process of consistency check rules involves the following steps:

- The primary TGG link contained in the correspondence component of the TGG production is set to "bound". The «create» annotation is removed from the primary TGG link.
- NACs are removed from the input component and the output component.
- «create» annotations are removed from all elements contained in the source and target component. This is due to the fact that all elements created by a TGG production—or by a translator—must be present in the context of the TGG link that was created by the TGG production.
- TGG parameters that are bound to attributes in both source and target production components are resolved either in source or target domain and the corresponding attribute is dropped. This is due to the fact that we demand that every operation that calculates an attribute value is invertible[8]. In general, the preference for resolving the TGG parameter either in source or target domain is equal. But, if a TGG parameter is used in a well-known operation in one of the components (e.g., operation "concat" in target component of $p^{(r)}$) then this domain is chosen. Otherwise, the TGG parameter is resolved in the source domain and the attribute in the target domain is dropped. Therefore, we have a derivation rule for TGG parameters that can be implemented deterministically.
- Attributes which are bound to a TGG parameter but have no corresponding attribute in the opposite domain are dropped. Such attributes are neither context elements of the TGG production nor do they have a corresponding attribute. Therefore, such attributes need not to be involved in the consistency check process.
- All assignment operators ":=" of attributes are set to the comparison operator "=".

The consistency check algorithm (cf. Listing 5) loops through all TGG links. For every TGG link in the graph triple the algorithm selects the CC rule derived from a TGG production

[7] Note that we have augmented the consistency check rules depicted in Fig. 6.10 with a stereotype that depicts the name of the classifying TGG link type of the primary TGG link because this information is not depicted due to the "bound" notation. The stereotype is placed near the primary TGG link.

[8] Otherwise, the TGG parameter would have to be resolved in both domains and none of the attribute assertions would be dropped.

that has created the TGG link. That is, we assume that every TGG link knows which TGG production (or derived translation rule) created the TGG link. Then the consistency check rule is executed and it checks whether the given TGG link is consistent. If this is not the case, the TGG link is added to a set of inconsistent TGG links. Operation "isConsistent(TGGLink)" corresponds to the derived CC rule and binds the given TGG link to the TGG link of the CC rule that has been set to "bound". Finally, the set of inconsistent TGG links is returned by the algorithm. If the set is empty then all TGG links are consistent according to their TGG rule specification. Otherwise the set contains all TGG links that are inconsistent according to their TGG rule specification.

```
1  procedure Set<TGGLink> checkTGGLinkConsistency(graphTriple: GraphTriple) {
2    inconsistentLinks: Set<TGGLink>;
3    tggLinkGraph: Graph = graphTriple.getCorrespondenceGraph();
4    forall (tggLinkNode ∈ tggLinkGraph) {
5      select ccRule derived from tggProduction that created tggLinkNode;
6      if (not ccRule.isConsistent(tggLinkNode)) {
7        inconsistentLinks.add(tggLinkNode);
8    } }
9    return inconsistentLinks;
10 }
```

Listing 5: Algorithm that checks consistency of TGG links.

Note that the consistency check algorithm in Listing 5 is only able to check the consistency of already existing TGG links. It is not able to detect new elements that have been added to the source or target domain that are not connected with a TGG link. Such new elements are untranslated, i.e., have an unmarked checkbox in our notation of elements. However, a check that searches the source and target domain for newly created elements that make the resulting graph triple inconsistent according to the TGG specification can be added easily in a separate algorithm. Moreover, the consistency check algorithm is not able to detect missing TGG links in the correspondence domain, i.e., TGG links that must be added to the graph triple to connect new elements that have been added to the source and target domain. The consistency check algorithm in Listing 5 has been designed to only check the consistency of a given set of TGG links according to the TGG specification. The algorithm can be used in a tool that visualizes TGG links to mark TGG links as inconsistent. Another feature of this algorithm is that it only marks TGG links as inconsistent that have "direct" inconsistent context elements in source, target, or correspondence domain. Inconsistencies in context elements of related TGG links are not propagated. For example, an inconsistency in a TGG link of TGG_{CDDS} that relates a class with a table would not propagate this inconsistency to TGG links that relate an attribute with a column. This way only "direct" inconsistencies of TGG links are reported and consecutive or inherited inconsistencies are suppressed.

7. Implementation of Approach

This chapter describes the implementation of the approach presented in this thesis. First, the MOFLON meta-CASE tool is presented in Sect. 7.1. Then, the implementation of the TGG components is discussed in Sect. 7.2. Finally, we discuss the tool integration framework, which is used to control the execution of integration scenarios, in Sect. 7.3. More detailed tutorials for MOFLON and the TGG editor can be found at `http://moflon.org`.

7.1. The MOFLON meta-CASE Tool

The concepts presented in this thesis are implemented in the meta-CASE tool MOFLON. MOFLON originated in 2003 at the Real-Time Systems Lab of the Technische Universität Darmstadt in order to provide MOF metamodeling support. For a description of the history of MOFLON we refer to [Kön09]. MOFLON [AKRS06] is based on the CASE tool Fujaba [NNZ00]. Fujaba provides an open plugin architecture and offers a GUI framework for diagram editors based on Java Swing. Fujaba supports creation of structural diagrams (UML 1.5-like class diagrams) and behavioral diagrams (story driven modeling diagrams) as well as code generation for both structural and behavioral diagrams. Plugins can (re)use and extend these structural and behavioral parts and the code generator. The current version of MOFLON uses Fujaba as basic editing platform, reuses some of its functionality, especially story driven modeling, and adds new functionality on top. To this end, MOFLON supports creation of MOF-compliant models, OCL expressions, story driven modeling, triple graph grammars and code generation for the provided models.

7.1.1. Architecture of MOFLON

Figure 7.1 depicts the architecture of MOFLON. MOFLON provides a MOF 2.0 editor, an SDM editor (which comes from Fujaba) as well as a TGG editor. These editors allow for the creation of MOF/OCL, SDM and TGG specifications. MOF/OCL specifications can also be imported by an XMI module from other CASE tools. The created specifications are given to a set of code generators and compilers which create JMI-compliant Java code from these specifications. The code generators utilized by MOFLON are: An XSLT based code generation engine called MOMoC [Bic04] which generates code for MOF models. Moreover, the Dresden OCL compiler toolkit[1] generates an implementation for OCL expressions. Finally, a code generation engine provided by Fujaba, which is modified by an alternative set of templates, is used to compile SDM diagrams into Java code.

[1] `http://www.dresden-ocl.org`

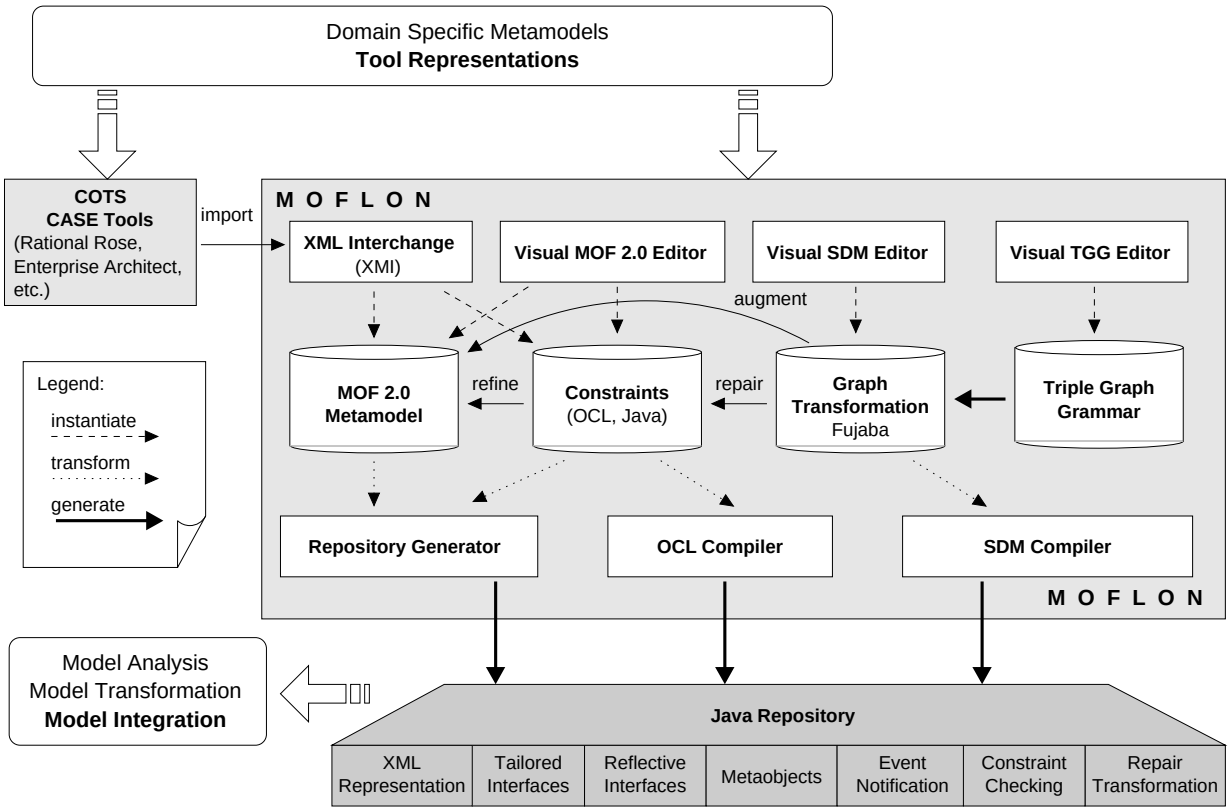


Figure 7.1.: Architecture of MOFLON (from [AKK+08]).

The generated code is bundled in a so-called (JMI) repository which provides a standardized set of interfaces, metaobjects, event notifications, and constraint checking operations. JMI is a standard which has been developed for MOF 1.4. Unfortunately, there exists no updated version of the JMI standard for MOF 2.0 up till now. Therefore, repositories generated by MOFLON are either based purely on JMI or on an extended set of interfaces derived from the JMI interfaces in order to provide certain concepts of MOF 2.0. For example, JMI provides a Java interface named *RefAssociationLink*, which is used to implement an instance of an association, i.e., a link. This interface simply provides methods for accessing the two members of the link, i.e., the two objects this link connects. MOFLON provides an interface *MOFLONAssociationLink* that extends *RefAssociationLink* and adds a new method "getClassifier" which returns the association that classifies the link as supposed by the CMOF Abstract Semantics of MOF 2.0 [Obj06].

A repository can be used by other programs or frameworks for model analysis, transformation or integration tasks. This thesis focuses on model integration tasks and the generated repositories are used by our integration framework as discussed in Fig. 3.10 (cf. Sect. 3.6).

7.1.2. MOFLON editors

Figure 7.2 depicts a screenshot of MOFLON. The editing pane is divided into a left- and a right-hand side. On the left-hand side there is a tree view which allows to select MOF and TGG projects. Moreover, it shows the elements contained in the packages belonging to a project. The right-hand side shows the diagrams of a project. In Fig. 7.2 two MOF diagrams have been opened that show the language models of class diagrams and database schemata (cf. Fig. 3.2 in Sect. 3.3.1) that have been specified as two MOF projects with MOFLON. The MOF editor uses a bootstrapped MOF 2.0 metamodel implementation and, therefore, supports all MOF 2.0 elements, e.g., packages, package imports and merges, classes, data types, primitive types, enumerations, associations, attributes, operations, constraints, and tags. Graph/model transformations are specified as SDM diagrams which are attached to certain operations.

Fujaba is based on a proprietary metamodel which provides an abstraction layer for model elements. This level of abstraction is provided as a set of interfaces, named F-interfaces. The Fujaba GUI framework requires these F-interfaces to be implemented by elements that should be visualized by the framework. Whenever changes occur in the model an event based mechanism automatically updates the GUI. Likewise, whenever changes occur in a graphical representation of a model element the underlying data structure is synchronized. Moreover, SDM diagrams require a structural model to be implemented as F-interfaces in order to function properly. Therefore, the MOFLON MOF-editor has implemented a so-called Fujaba-MOF-Adapter layer to connect the bootstrapped MOF repository to the F-interfaces. This adapter is hard to maintain if changes occur in either the MOF implementation or in the F-interfaces. The integration into the Fujaba GUI framework is even more difficult. Unfortunately, Fujaba's GUI framework does not provide certain features typical for modern GUI frameworks, as zooming and auto-routing. Consequently, the MOFLON architecture is currently changed such that other GUI frameworks are usable by MOFLON in future versions.

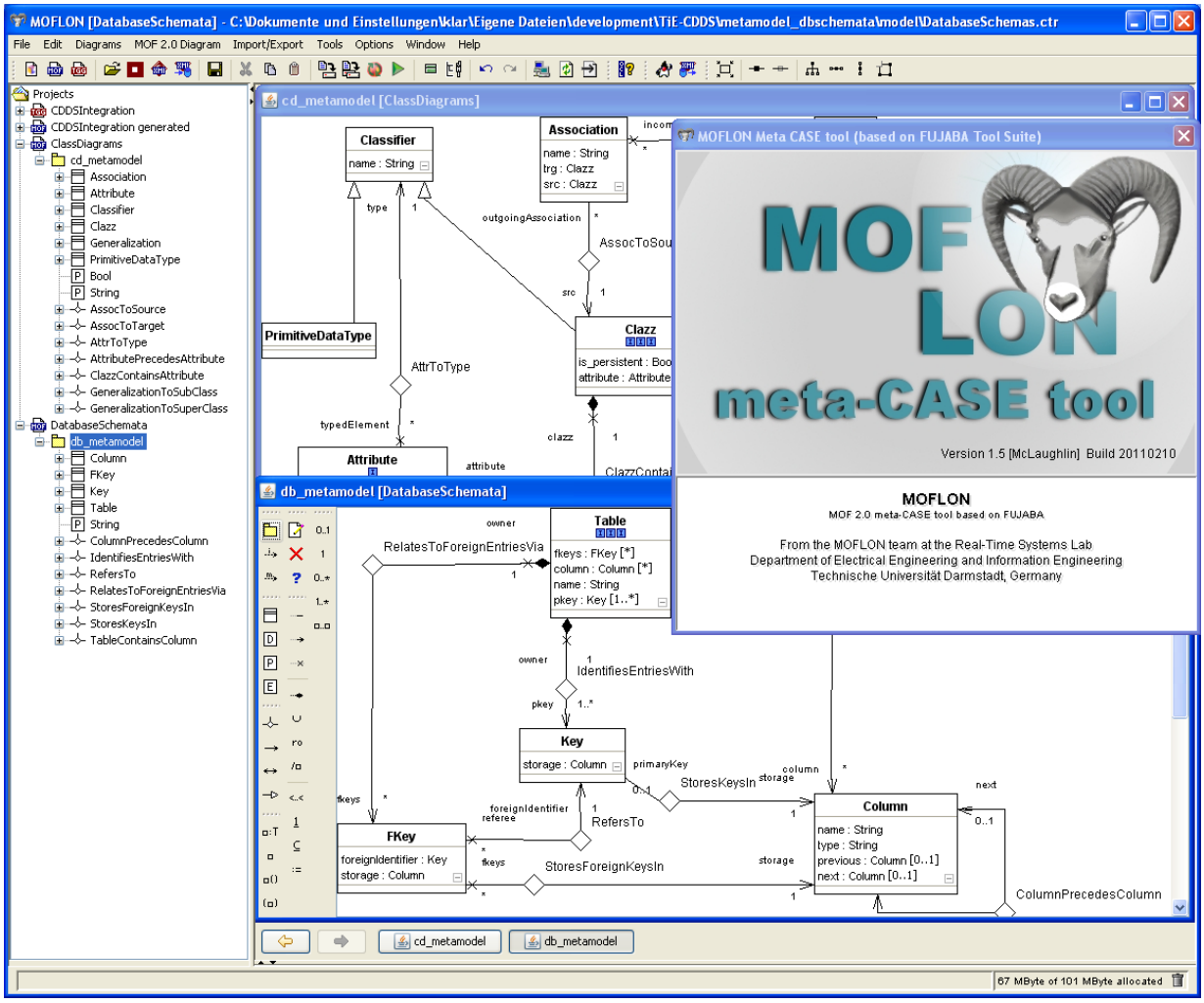


Figure 7.2.: Screenshot of MOFLON.

Fujaba's editing framework provides a transaction based persistence support mechanism. Every time a model element is changed the change is persisted, i.e. deltas that occur in a model are persisted. During development this produced many problems in persisted files when errors occurred while a change was in progress. Consequently, unfinished transactions were rolled back but during the rollback further errors occurred which leaded to inconsistent (i.e., corrupted) model states.

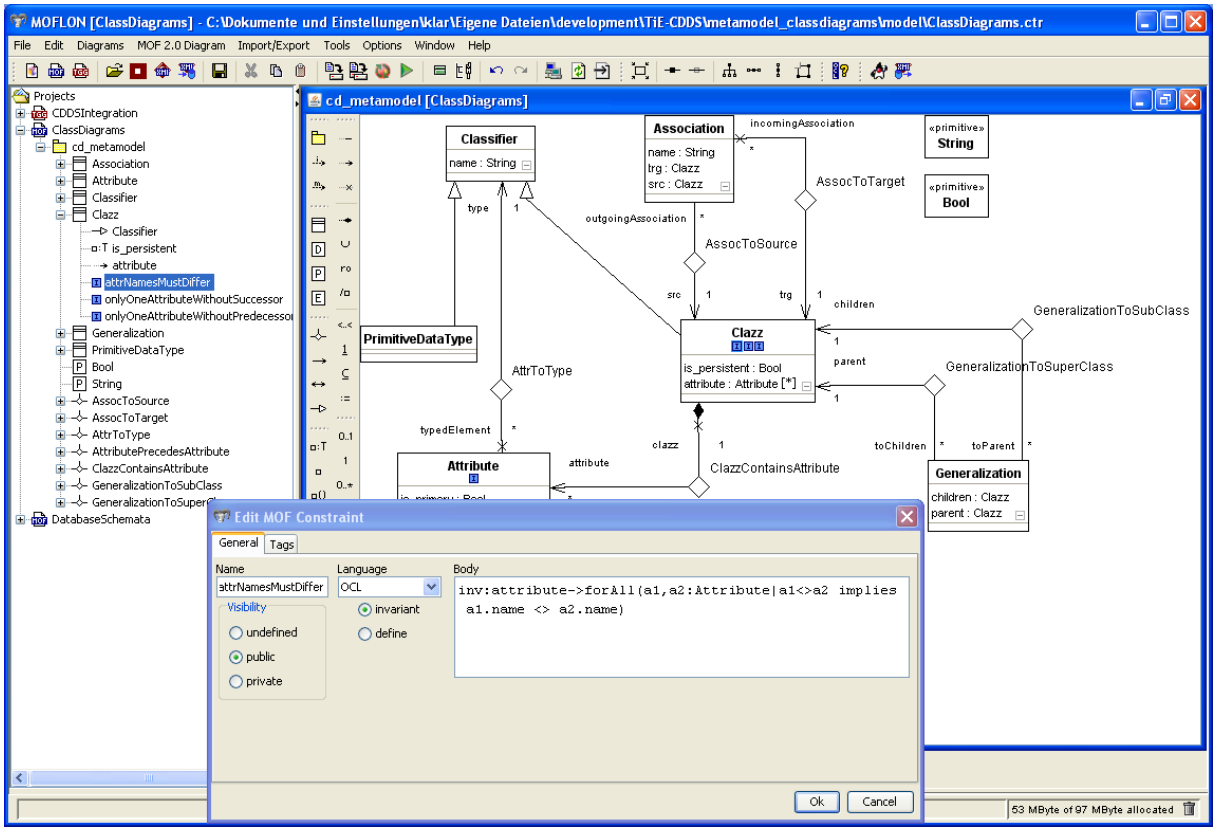


Figure 7.3.: MOFLON's OCL expression editor.

MOFLON provides support for enriching certain MOF elements with OCL expressions (cf. Fig. 7.3). During code generation, OCL expressions are passed to the Dresden OCL compiler toolkit which generates Java code from these expressions. The code generated by the OCL compiler is then integrated with the code generated by the repository generator. Figure 7.3 depicts the OCL invariant attached to the context of a class that demands that every two attributes owned by a class must have different names (cf. Sect. 3.3.2). During runtime a distinguished operation can be performed which checks whether this constraint is fulfilled.

In addition, the repository generator adds multiplicity constraint checks for every association end. This enables to check at repository runtime whether the multiplicity constraints of an association end (e.g., "1..*") are fulfilled.

7.2. TGG in MOFLON

Figure 7.4 depicts the architecture of the workflow of MOFLON's TGG editor. The TGG editor consists of a TGG schema editor, a TGG rule editor[2], a generator, and a set of rule derivation strategies. The schema editor requires a source and target metamodel to which it connects the specified TGG link types. The rule editor depends on the schema editor because each TGG production is attached to a certain TGG link type. Both TGG schema and rule specification are given as TGG project to the generator that translates this TGG project into a MOF and SDM representation. The resulting MOF specification contains the structural part of the TGG project, whereas the behavioral part (i.e., the operational rules) is specified as SDM diagrams. The resulting MOFLON MOF project is then passed to MOFLON's generator and compiler backend (cf. Fig 7.1) which results in JMI compliant integration rule code. This integration rule code corresponds to the two components *generated translation rule code* and *generated repository of link management tool adapter* depicted in Fig. 3.10 (cf. Sect. 3.6). The integration framework, i.e., the integrator, then uses this code to perform certain integration tasks in the integration scenario it is currently executing.

7.2.1. TGG Editor

The TGG editor is implemented as Fujaba plugin. As a structural editor, the TGG schema editor implements certain structural F-interfaces. Contrary, the TGG rule editor is a behavioral editor, which has been designed as a specialization of the SDM implementation. It extends certain SDM datatypes and, in addition, the rule editor is aligned with the SDM editor. The implementation of the TGG editor is based upon two metamodels. One for the schema editor and one for the rule editor. The description of these metamodels is out of scope for this contribution. The interested reader is referred to [Kön09] where these metamodels are explained in detail.

A screenshot of the TGG schema editor is depicted in Fig. 7.5. The TGG schema editor allows to create TGG link types and to structure them in TGG packages. The outermost TGG package *cdds* attaches to the packages that contain the language models of class diagrams and database schemata, which allows to integrate the elements of both language models in this TGG package. Five TGG link types have been specified in Fig. 7.5. Note that due to technical reasons association *Inherits* has been "normalized" to a class named *Generalization* because instances of an association, i.e., links, cannot be treated as first class members in the current story driven modeling implementation, e.g., no connections can be attached to links. The depicted TGG link types are the realization of the TGG schema depicted in Fig. 4.3 (cf. Sect. 4.2). To each of the TGG link types a TGG production is attached. Note that the multiplicities of the ends of TGG link types have been set to either "1" or "0..1". The semantics is as follows: If the multiplicity of an end of a TGG link type, e.g., type *Class*, is set to "1", then for each instance of the opposite end, e.g., type *Table*, an instance of a TGG link type *must exist* that connects the opposite instance with an instance of the other end of the TGG link type. If the multiplicity of an end of a TGG link type, e.g., type *Attribute*, is set

[2]Due to historical reasons it is *TGG rule editor* instead of *TGG production editor*.

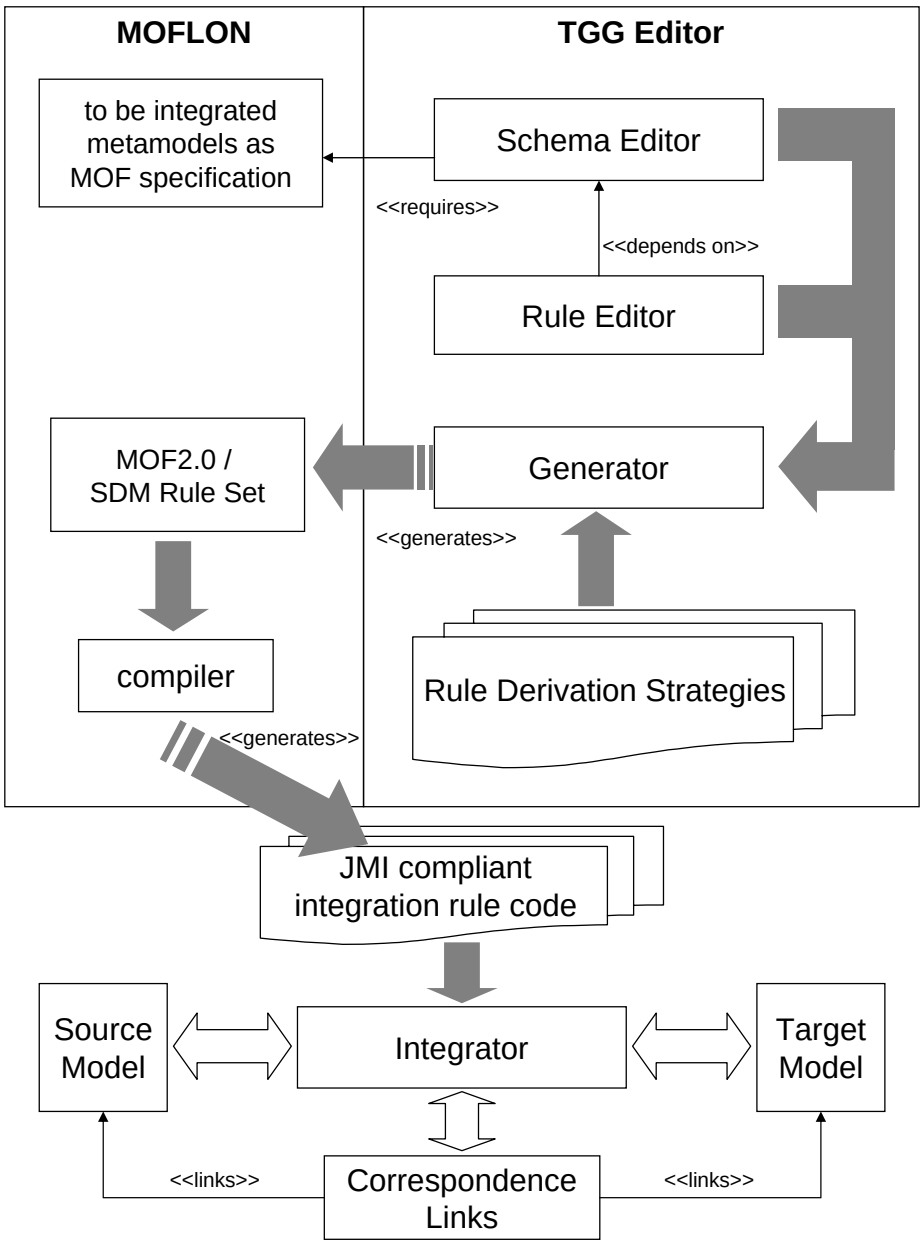


Figure 7.4.: Architecture of the TGG-Editor (from [KKS07]).

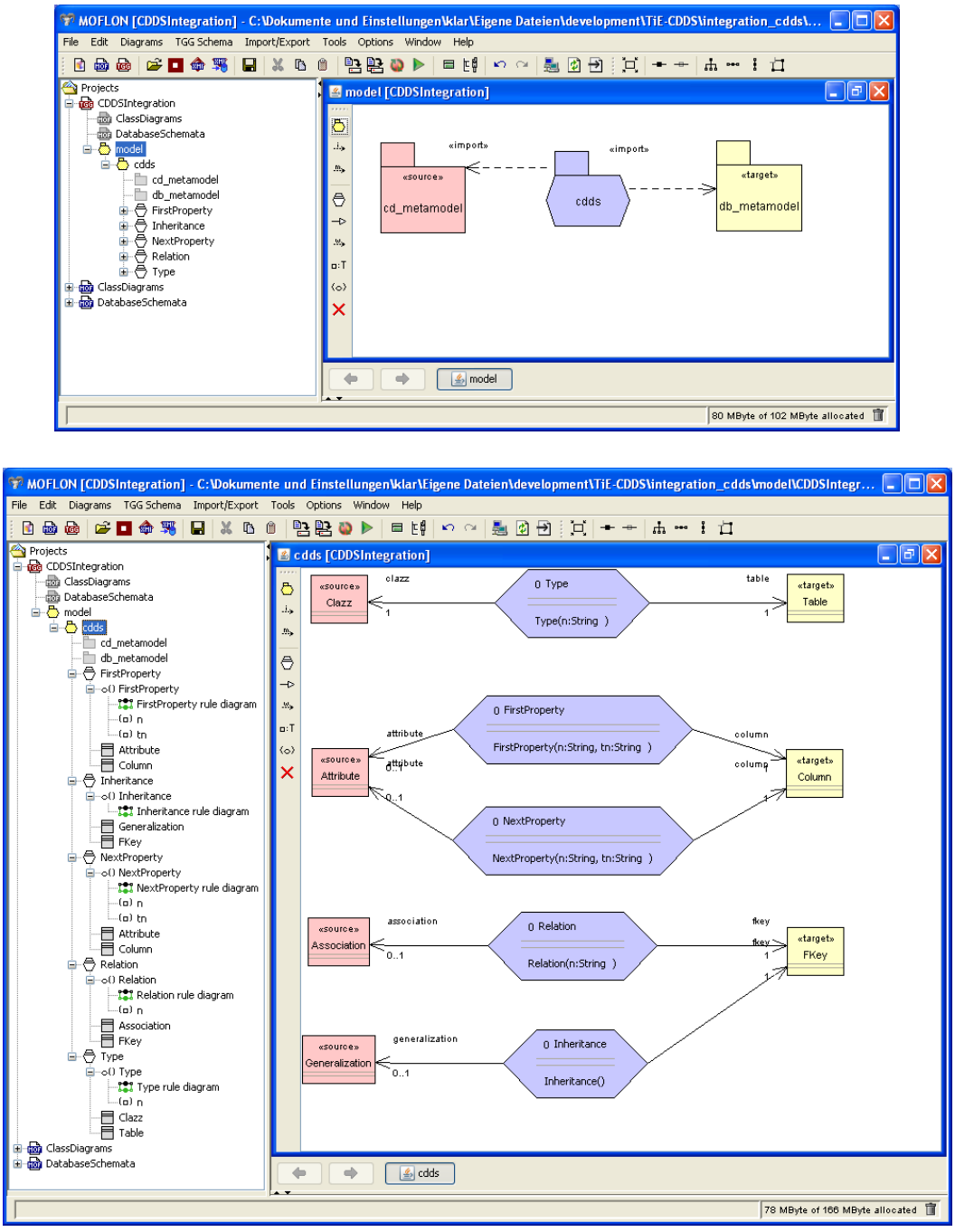


Figure 7.5.: TGG schema editor: packages and link types.

to "0..1", then for each instance of the opposite end, e.g., type *Column*, an instance of a TGG link type *may—but need not—exist* that connects the opposite instance with an instance of the other end of the TGG link type. The rationale for this is rather intuitive: for each class there must exist a corresponding table and for each table there must exist a corresponding class. Furthermore, for each attribute there must exist a corresponding column. But, for each column there need not exist a corresponding class because according to the TGG productions of TGG_{CDDS} (cf. Fig. 4.6 in Sect. 4.4) there are some columns that are used as secondary elements which do not have a corresponding element in the opposite domain.

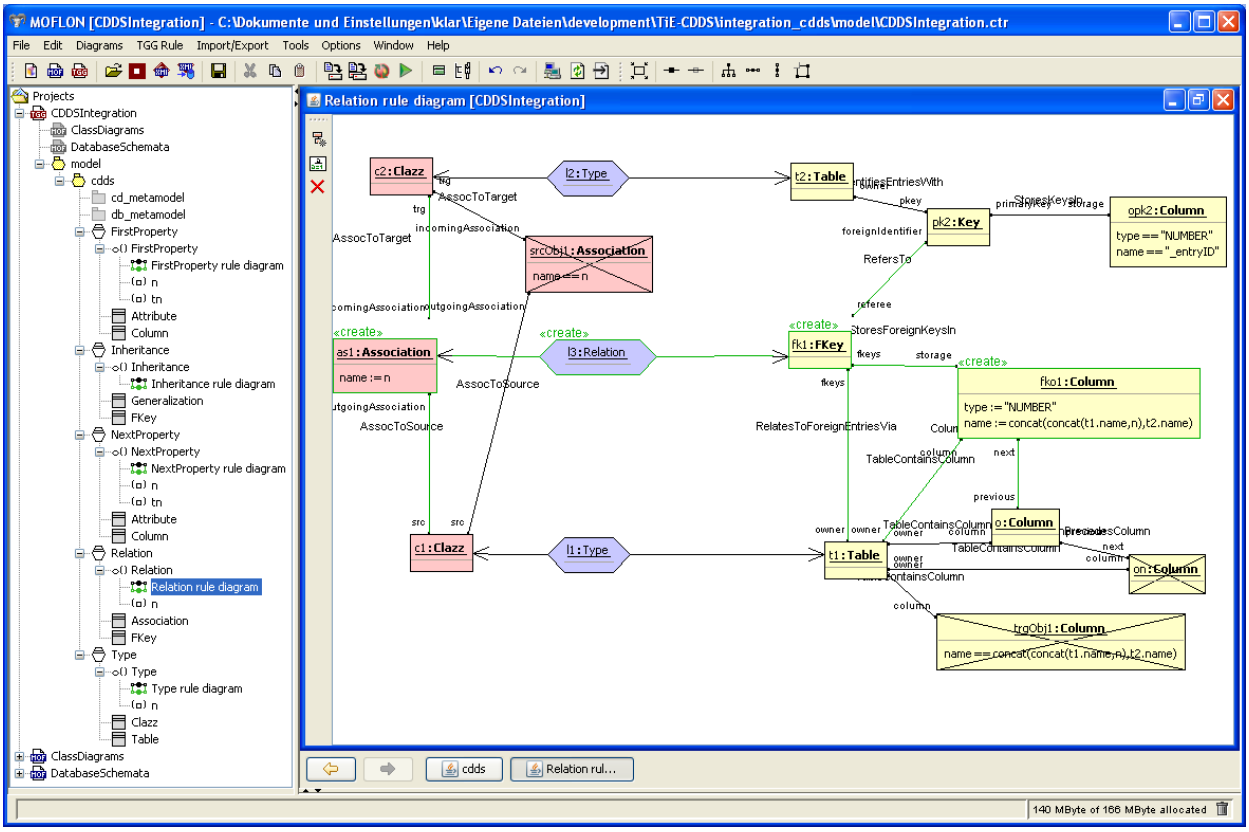


Figure 7.6.: TGG rule editor: TGG productions.

TGG productions are edited in the TGG rule editor. A screenshot that shows the TGG production associated with TGG link type *Relation* (cf. Fig. 4.6) is depicted in Fig. 7.6. As already mentioned, the rule editor is a behavioral editor that allows to model declarative TGG productions. Figure 7.6 contains a pattern consisting of certain (negative) objects, links, and TGG links (cf. Sects. 4.3 and 4.4 for a detailed discussion of TGG productions).

7.2.2. Translating a TGG Project

A TGG project is translated into a MOFLON MOF project in order to generate code from it. The MOF project is an intermediate modeling artifact and is used for debugging purposes only.

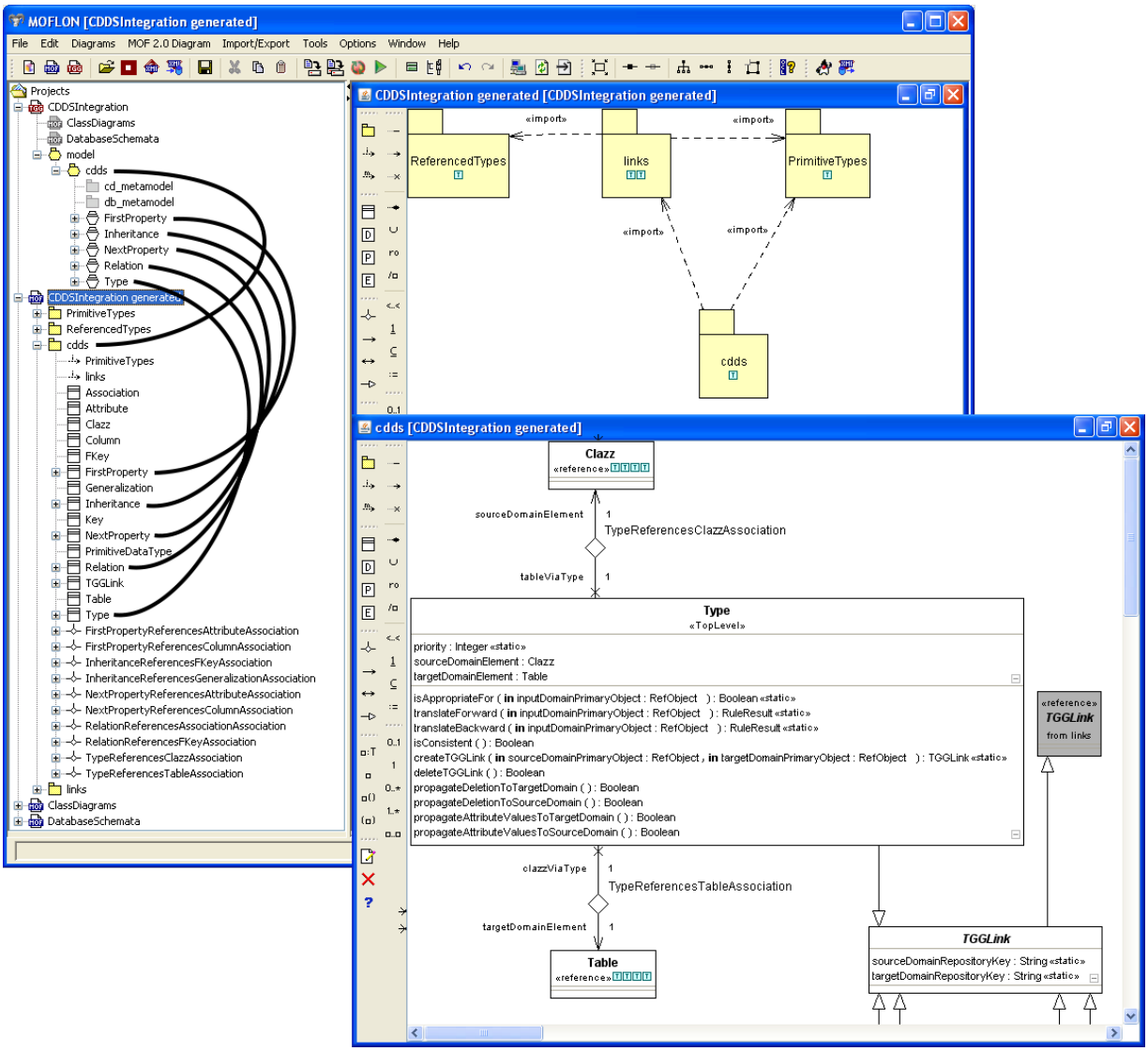


Figure 7.7.: MOF project generated from TGG project.

Figure 7.7 depicts the MOF project generated from the TGG schema depicted in Fig. 7.5 and the operational rules generated from the TGG productions of TGG_{CDDS}. Besides two technical packages containing referenced types and primitive types, a package named "links" containing a part of the TGG links runtime metamodel is generated. However, from these packages no code will be generated because the code still exists in a library and is accessed from the generated code. In addition, for each TGG package a MOF package is created. For these packages code is generated later on. Then, each TGG link type is mapped to a MOF class that derives from the class *TGGLink*. The mapped class is connected via two associations to the elements of the source and the target domain respectively. The TGG elements in the TGG project and their corresponding elements in the MOF project have been marked in Fig. 7.7.

Class *TGGLink* defines an interface that must be implemented by every deriving TGG link so the integration framework is able to execute TGG algorithms successfully. Altogether, quite a number of operational rules that are derived from a TGG production are part of this interface. However, most of them are used to perform incremental update operations which are out of scope in this contribution. The operational rules which are relevant for this contribution are:

- "isAppropriateFor(RefObject): Boolean": This operation checks whether the given object is type compatible with the primary element of source and target domain respectively. This operation is used by TGG algorithms in order to determine the candidate rules for translating a given primary object.
- "translateForward(RefObject): RuleResult": This operation translates the given primary element of the source domain into a corresponding element structure of the target domain according to the specification in the TGG production. The return object of type *RuleResult* is part of the TGG links runtime metamodel and contains the following information about the translation process:
  - whether the translation process was successful,
  - which elements are required as context elements by the translation process,
  - which secondary elements are included in the translation process,
  - which elements where created in the opposite domain, i.e., target domain, and
  - a pointer to the TGG link created by this operation.
- "translateBackward(RefObject): RuleResult": Same as operation "translateForward" but for translating in the reverse direction, i.e., from target domain to source domain.
- "isConsistent(): Boolean": This operation checks whether an instance of this TGG link is consistent with the situation specified in the TGG production.

Figure 7.8 depicts operational rule "translateForward" derived from the TGG link type *Type* and its corresponding TGG production. This rule is used by TGG algorithms during forward translation. The rule derivation strategy applied to generate this operational rule is inspired by the derivation process of forward/backward translation rules from a TGG production described

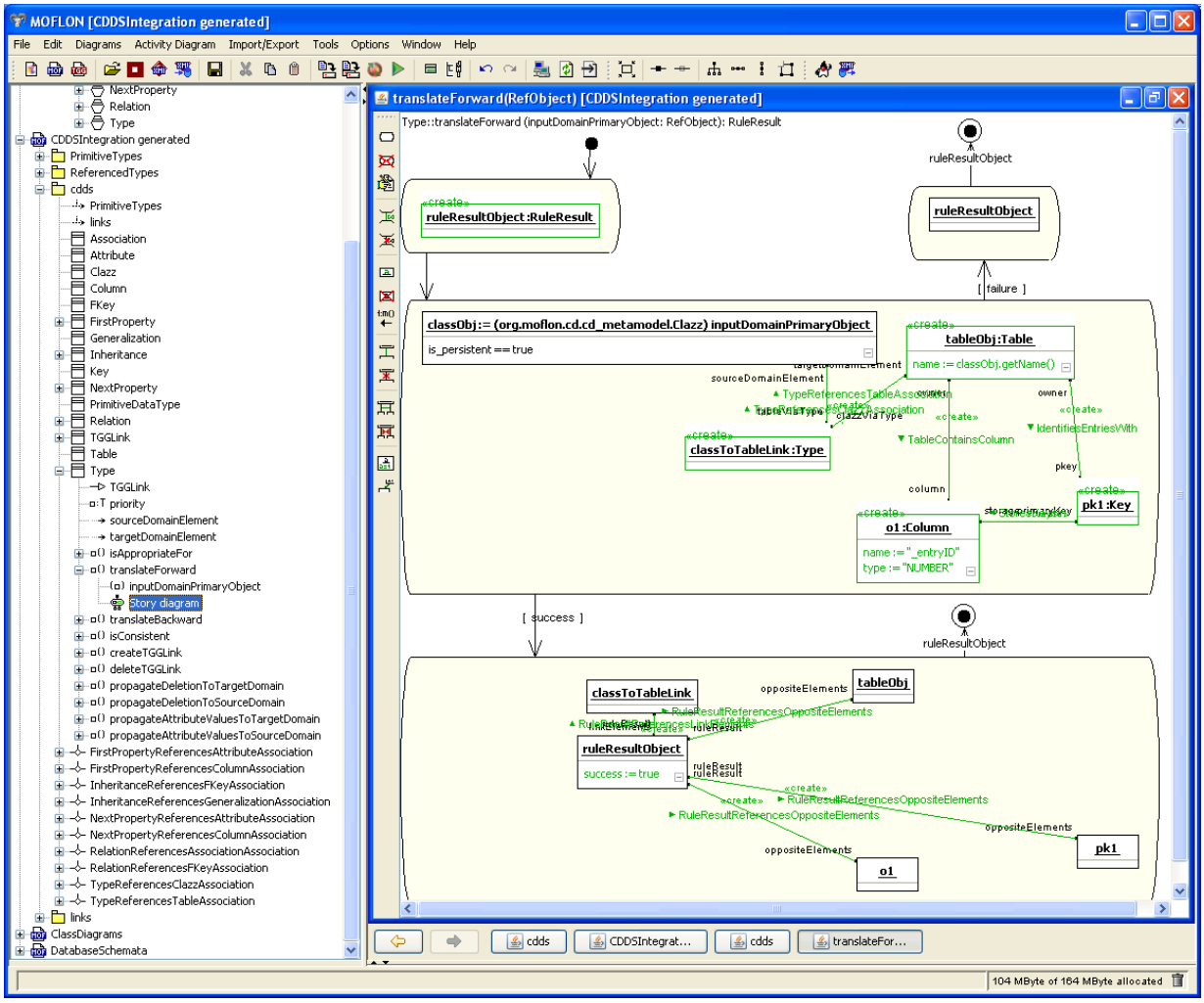


Figure 7.8.: Operational rule *translateForward* generated from TGG production.

in Sect. 5.2.3. The operational rule "translateForward" is divided into four activities. First, an instance of type *RuleResult* is created. Then, the main operation that tries to translate a given class element into a corresponding table structure is executed. A TGG link of type *Type* is then created that connects the given class element with the newly created table element. If this operation fails then the rule result object is simply returned stating that the operation was not successful. In the general case, i.e., when context elements are matched by a translation rule, these context elements are also added to the set of required elements managed by the rule result object when the operation fails. Moreover, additional secondary elements, i.e., elements to-be-translated by the rule, are also added to the set of of included elements managed by the rule result object. Otherwise, if the main operation succeeds, then the rule result object is modified, such that the newly created elements in the target domain are added to the collection of opposite elements. Moreover, the newly created TGG link is added to the collection of TGG links managed by the rule result object. As the TGG production does not create additional secondary elements in the source domain no elements need to be added to the collection of included elements managed by the rule result object. Likewise, no additional context elements are required to be translated beforehand, so the collection of required elements managed by the rule result object does not need to be modified.

The rule result object manages all elements touched by the translation rule. Note that *core rules* (cf. Sect. 6.2) are used by the algorithm in Listing 4 (cf. Sect. 6.6). These core rules are used to identify the core matches of a translation rule. The core rules of all TGG productions of TGG_{CDDS} used by our algorithm have been presented in Fig. 6.1 and are already discussed in Sect. 6.2. They look very similar to the operational forward and backward operations. Moreover, they are almost identical despite the fact that only elements of the input domain are part of a core rule. So, core rules and operational translation rules use the same rule result structure to identify elements that are used as context elements or that are secondary elements to-be-translated by the corresponding translation rule. The elements matched by a core rule are passed to the algorithm via the rule result object. So, the algorithm in Listing 4 operates on elements that are managed by the rule result object after a call to a core rule.

Figures 7.9 and 7.10 depict the Java code that corresponds to the operational rule depicted in Fig. 7.8. The four activities belonging to operation "translateForward" that contain story patterns have been highlighted in Figs. 7.9 and 7.10 for better readability. The Java code is generated with the SDM compiler CodeGen2. The SDM compiler uses a special set of templates that is able to establish a connection to a repository registry implementation. The repository registry is used every time an element from source, link, or target domain is queried or created (cf. Sect. 7.3.1 that explains why a registry is necessary in this case). In line 205 the repository registry is used to create an instance of a rule result element. This rule result element is used to mark distinguished created elements or elements used as context in the input domain. The call in line 225 creates a TGG link of type *Type* in the link domain. Lines 228 to 235 create three objects in the target domain—a table, a column, and a primary key. These objects are then linked with each other and with the TGG link in lines 250 to 281. Lines 312 to 328 update the rule result object. Therefore, the created TGG link and the created objects contained in the opposite domain are added to distinguished sets of the rule result object. Moreover, elements that are required as context elements before the translation operation is possible and additional secondary elements of the input domain are

TypeClassImpl.java

```
// generating operation translateForward (opWithNamespace=false)
public org.moflon.tgg.language.links.RuleResult translateForward(javax.jmi.reflect.RefObject inputDomainPrimaryObject) throws javax.jmi.reflect.JmiException
{
   boolean fujaba__Success = false;

   Object fujaba__templ = null;
   boolean fujaba__cond = false;
   int fujaba__index = 0;
   org.moflon.tgg.language.links.RuleResult ruleResultObject = null;
   org.moflon.cd.cd_metamodel.Clazz classObj = null;
   org.moflon.tgg.model.cdds.Type classToTableLink = null;
   org.moflon.ds.db_metamodel.Table tableObj = null;
   org.moflon.ds.db_metamodel.Key pk1 = null;
   org.moflon.ds.db_metamodel.Column ol = null;

   // story pattern
   try
   {
      fujaba__Success = false;

      // create object

      // try to locate type of instance in any registered repository
      ruleResultObject = (org.moflon.tgg.language.links.RuleResult) (org.moflon.facility.RepositoryRegistry.get().getInstance((RefPackage) null, "links.RuleResult"));
      fujaba__Success = true;
   } catch (JavaSDMException fujaba__InternalException)
   {
      fujaba__Success = false;
   }

   // story pattern
   try
   {
      fujaba__Success = false;

      classObj = (org.moflon.cd.cd_metamodel.Clazz) inputDomainPrimaryObject;
      // check object classObj is really bound: false
      JavaSDM.ensure(classObj != null);

      // attribute condition
      JavaSDM.ensure(classObj.isPersistent() == true);

      // create object
      classToTableLink = (org.moflon.tgg.model.cdds.Type) (org.moflon.facility.RepositoryRegistry.get().getInstance(refOutermostPackage(), "cdds.Type"));

      // create object
      tableObj = (org.moflon.ds.db_metamodel.Table) (org.moflon.facility.RepositoryRegistry.get().getInstance(
            (String) refGetValue("targetDomainRepositoryKey"), "db_metamodel.Table"));

      // create object
      pk1 = (org.moflon.ds.db_metamodel.Key) (org.moflon.facility.RepositoryRegistry.get().getInstance((String) refGetValue("targetDomainRepositoryKey"), "db_metamodel.Key"));

      // create object
      ol = (org.moflon.ds.db_metamodel.Column) (org.moflon.facility.RepositoryRegistry.get().getInstance((String) refGetValue("targetDomainRepositoryKey"), "db_metamodel.Column"));

      // assign statement
      tableObj.setName(classObj.getName());

      // assign statement
      ol.setName("_entryID");

      // assign statement
      ol.setType("NUMBER");

      // create to-one link
      classToTableLink.setSourceDomainElement(classObj);

      // create to-one link
      fujaba__templ = ((org.moflon.ds.db_metamodel.IdentifiesEntriesWith) (org.moflon.facility.RepositoryRegistry.get().getAssociation(
            (String) refGetValue("targetDomainRepositoryKey"), "db_metamodel.IdentifiesEntriesWith"))).getOwner(pk1);

      if (fujaba__templ != null)
      {
         ((org.moflon.ds.db_metamodel.IdentifiesEntriesWith) (org.moflon.facility.RepositoryRegistry.get().getAssociation(
               (String) refGetValue("targetDomainRepositoryKey"), "db_metamodel.IdentifiesEntriesWith"))).remove(
               (org.moflon.ds.db_metamodel.Table) fujaba__templ, pk1);
      }

      ((org.moflon.ds.db_metamodel.IdentifiesEntriesWith) (org.moflon.facility.RepositoryRegistry.get().getAssociation(
            (String) refGetValue("targetDomainRepositoryKey"), "db_metamodel.IdentifiesEntriesWith"))).add(tableObj, pk1);

      // create to-one link
      fujaba__templ = ((org.moflon.ds.db_metamodel.TableContainsColumn) (org.moflon.facility.RepositoryRegistry.get().getAssociation(
            (String) refGetValue("targetDomainRepositoryKey"), "db_metamodel.TableContainsColumn"))).getOwner(ol);

      if (fujaba__templ != null)
      {
         ((org.moflon.ds.db_metamodel.TableContainsColumn) (org.moflon.facility.RepositoryRegistry.get().getAssociation(
               (String) refGetValue("targetDomainRepositoryKey"), "db_metamodel.TableContainsColumn"))).remove(
               (org.moflon.ds.db_metamodel.Table) fujaba__templ, ol);
      }

      ((org.moflon.ds.db_metamodel.TableContainsColumn) (org.moflon.facility.RepositoryRegistry.get().getAssociation(
            (String) refGetValue("targetDomainRepositoryKey"), "db_metamodel.TableContainsColumn"))).add(tableObj, ol);

      // create to-one link
      classToTableLink.setTargetDomainElement(tableObj);

      // create to-one link
      pk1.setStorage(ol);

      fujaba__Success = true;
   } catch (JavaSDMException fujaba__InternalException)
   {
      fujaba__Success = false;
   }
```

Figure 7.9.: Java code generated for operational rule *translateForward* (part 1 of 2).

```
289     if (fujaba__Success)
290     {
291        // story pattern
292        try
293        {
294           fujaba__Success = false;
295
296           // check object classToTableLink is really bound: false
297           JavaSDM.ensure(classToTableLink != null);
298
299           // check object ol is really bound: false
300           JavaSDM.ensure(ol != null);
301
302           // check object pkl is really bound: false
303           JavaSDM.ensure(pkl != null);
304
305           // check object ruleResultObject is really bound: false
306           JavaSDM.ensure(ruleResultObject != null);
307
308           // check object tableObj is really bound: false
309           JavaSDM.ensure(tableObj != null);
310
311           // assign statement
312           ruleResultObject.setSuccess(true);
313
314           // create to-many link
315           ((org.moflon.tgg.language.links.RuleResultReferencesOppositeElements) (org.moflon.facility.RepositoryRegistry.get().getAssociation(
316                 (RefPackage) null, "links.RuleResultReferencesOppositeElements"))).getRuleResult(tableObj).add(ruleResultObject);
317
318           // create to-many link
319           ((org.moflon.tgg.language.links.RuleResultReferencesTggLinks) (org.moflon.facility.RepositoryRegistry.get().getAssociation((RefPackage) null,
320                 "links.RuleResultReferencesTggLinks"))).getRuleResult(classToTableLink).add(ruleResultObject);
321
322           // create to-many link
323           ((org.moflon.tgg.language.links.RuleResultReferencesOppositeElements) (org.moflon.facility.RepositoryRegistry.get().getAssociation(
324                 (RefPackage) null, "links.RuleResultReferencesOppositeElements"))).getRuleResult(ol).add(ruleResultObject);
325
326           // create to-many link
327           ((org.moflon.tgg.language.links.RuleResultReferencesOppositeElements) (org.moflon.facility.RepositoryRegistry.get().getAssociation(
328                 (RefPackage) null, "links.RuleResultReferencesOppositeElements"))).getRuleResult(pkl).add(ruleResultObject);
329
330           fujaba__Success = true;
331        } catch (JavaSDMException fujaba__InternalException)
332        {
333           fujaba__Success = false;
334        }
335
336        return ruleResultObject;
337     } else
338     {
339        // story pattern
340        try
341        {
342           fujaba__Success = false;
343
344           // check object ruleResultObject is really bound: false
345           JavaSDM.ensure(ruleResultObject != null);
346
347           fujaba__Success = true;
348        } catch (JavaSDMException fujaba__InternalException)
349        {
350           fujaba__Success = false;
351        }
352
353        return ruleResultObject;
354     }
355  } // end of method
```

Figure 7.10.: Java code generated for operational rule *translateForward* (part 2 of 2).

added to the rule result object if they exist in the operational rule. In this example there are no such elements. However, the backward translation operation of TGG link type *Type* would add the column and the primary key as additional secondary elements to the rule result object. Moreover, the code generated for the operational rules derived from the other TGG productions "FirstProperty", "NextProperty", "Relation", and "InheritanceRelation" would add required context elements to the rule result object (cf. Figs. 5.6 and 5.7 in Sect. 5.2.3 for the operational rules derived from the TGG productions).

7.3. Tool Integration Framework

The *tool integration framework* is used to instantiate three repositories: one for the source domain, one for the target domain, and one for the link domain that integrates source and target domain. Moreover, it is used to control operations that are performed on these repositories. An overview of the architecture of the integration framework has already been given in Sect. 3.6.

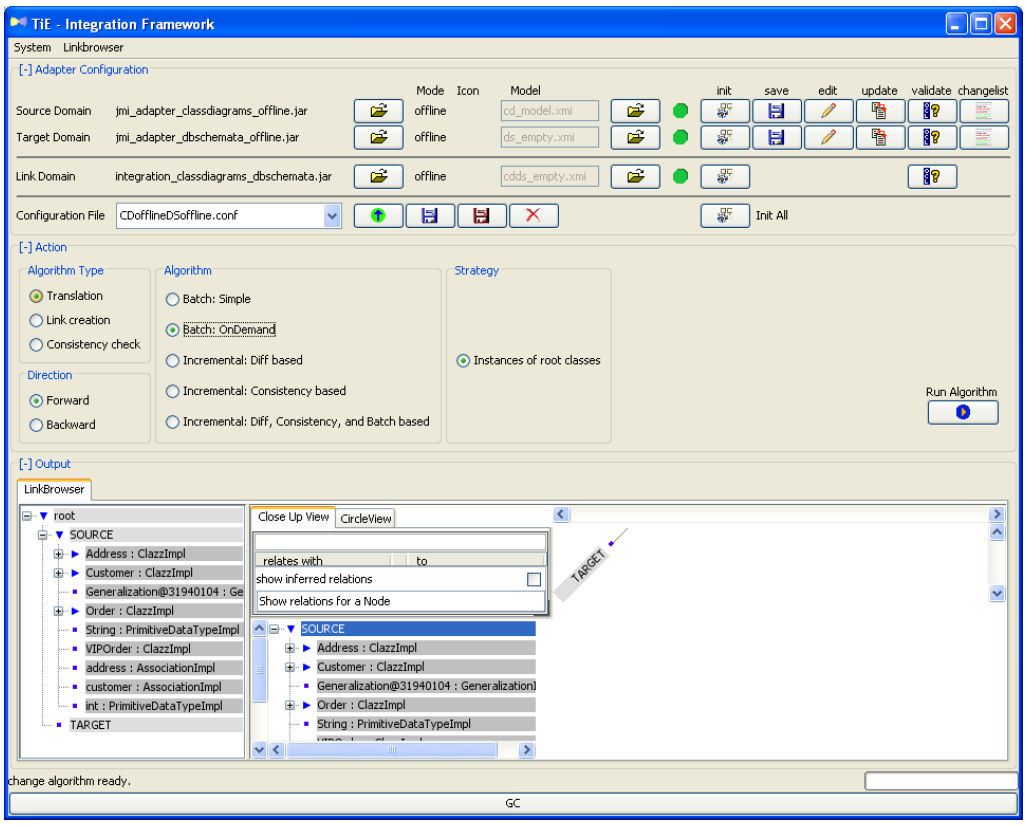


Figure 7.11.: Screenshot of the Integration Framework.

The integration framework (depicted in Fig. 7.11) provides a graphical user interface which is divided into three sections. The first section allows for the configuration of a TGG scenario. In our case, the class diagrams to database schemata integration scenario has been chosen.

The second section offers the possibility to configure an algorithm that performs certain operations on the three repositories. The third section provides a graphical view onto the three repositories. This view is realized by a *linkbrowser* component, which is a modified version of a software component named *matrix browser* [ZKB02]. After the repositories have been instantiated and an operation has been executed (cf. Fig. 7.12), the linkbrowser is refreshed and the interrelations between corresponding elements are visualized. On the left-hand side of the linkbrowser the elements of the source domain are visualized in a column. In the upper part of the linkbrowser the elements of the target domain are visualized in a row. The TGG links that connect elements from the source domain with elements from the target domain are visualized in between as entries of the matrix made up by the entries of the row and column.

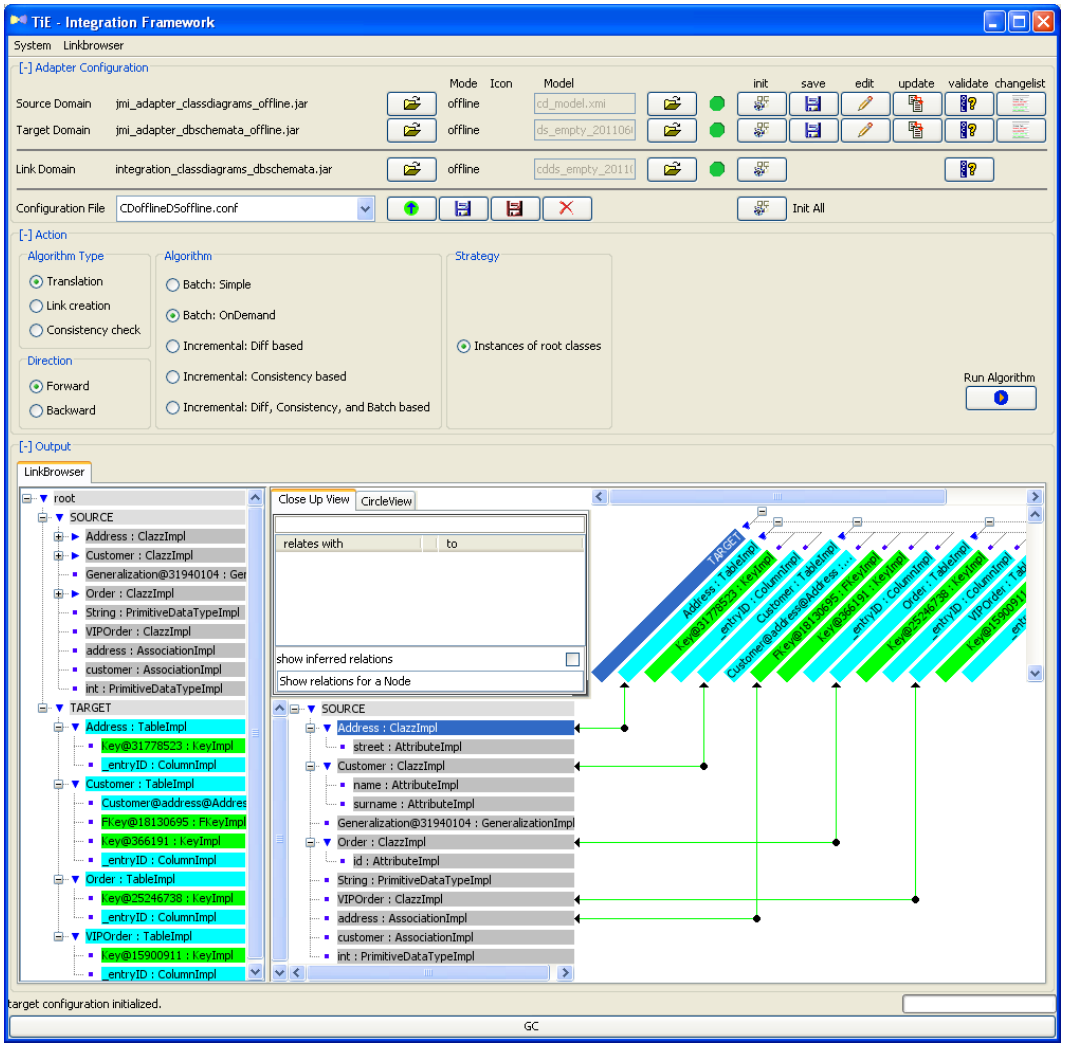


Figure 7.12.: Integration Framework after forward translation.

The linkbrowser marks elements with different colors:

- black: No changes where made to this element recently.

- green: The element has been created recently.
- red: The element has been deleted recently.
- cyan: A property of this element has been changed recently.

The linkbrowser allows to collapse elements that are in a certain hierarchy relation. For example, attributes are modeled as part of a composite relation to classes. Consequently, attributes are visualized beneath classes and can be folded.

Unfortunately, the linkbrowser visualization does not scale well if large models containing more than 1000 elements are displayed.

The implementation of the integration framework is divided into model, view, and controller according to the MVC pattern [GHJV95]. Consequently, the GUI is divided from the logic. TGG algorithms are stored in a separate module. However, different TGG algorithm implementations may be configured in the integration framework as a TGG algorithm implementation may have certain variation points.

The implementation of TGG links has an in-memory representation according to the JMI repository generated from the TGG project. However, serialization and visualization of TGG links is realized with two maps that are synchronized with the TGG link in-memory representation by using the provided event mechanisms. The maps are ID based as every element of the source and target domain has a unique ID. The connection of a TGG link to its source and target domain element is reflected by the existence of an entry in both maps.

The integration framework allows to check constraints during runtime of a repository, either source, link, or target repository. It simply calls the according operation "refVerifyConstraints(boolean deepVerify): Collection<javax.jmi.reflect.JmiException>" provided by the JMI standard. If this operation is called on the outermost package of a repository (with the parameter "deepVerify" set to true) then the whole repository is checked for fulfilling the specified constraints, also OCL constraints.

7.3.1. Accessing Repositories

The code generated for operational rules resides in the link domain repository. It needs to get access to elements that reside in foreign repositories, namely the source and the target domain repository. This is due to the fact that operational rules create, modify, and analyze elements in the source and target domain.

The problem is that MOF compliant implementations do not have the possibility to get outside of their repository. That is, the scope of a repository is limited to this repository and other repositories are not visible. For example, the repository of the link domain cannot access the source and target domain repositories directly. The OMG provides a solution for this insufficiency. It proposes a facility mechanism that is able to provide an entry connection point to the metadata contained in a repository [Obj10a]. A mechanism that is based on the ideas presented in this proposal has been implemented in MOFLON. Figure 7.13 depicts a class diagram of the vital part of the repository registry that realizes this mechanism. With this registry, the repository containing the translation code is able to access all necessary repositories: the source, target, and link domain repositories.

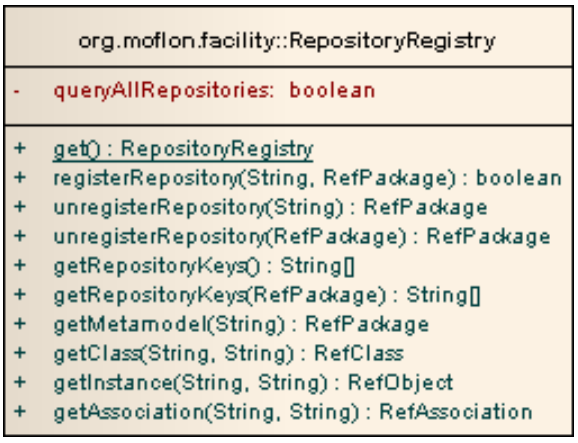


Figure 7.13.: Class diagram of the repository registry.

This registry is used for getting access to types[3] defined by the metamodel of a JMI compliant repository. The registry mainly allows to query types and to create instances of classes. An outermost package of a metamodel contained in a repository may be registered with a unique key. Typically, a metamodel contains exactly one outermost package, but in general is not restricted to only have one outermost package. Starting at the outermost package of a metamodel all nested elements can be queried by name using reflection. The registry implementation can also be used to search for a type in all registered outermost packages. However, there may be collisions if the type is available in more than one outermost package.

In terms of the MOF 2.0 Facility and Object Lifecycle Specification (MOFFOL) [Obj10a] this registry can be compared to a "Facility". A Facility contains zero to many Extents and represents the means for clients to "connect" to models and elements, which are accessed via Extents. An "Extent" is the MOF2 equivalent of the JMI "RefPackage" in MOF1.4. A Repository is the equivalent of the MOF2 concept "Extent". The Extents supported by this registry are "outermost packages". The registry uses unique keys associated to Extents instead of Extent::name to identify an Extent in this Facility.

The following modifications had to be done in the implementation of MOFLON. MOMoC was adjusted such that during creation time of a repository, a key is dynamically associated with this repository at runtime, i.e., a unique key is generated and used to register the repository at the facility. The TGG generator was adjusted such that each link repository, containing the TGG implementation, stores the keys of its source and target repository and provides a mechanism for getting access to this piece of information. The generator for SDMs that produces code for accessing elements of a foreign repository had to be adjusted, such that the code generated from operational rules uses the registry mechanism passing source and target keys to the registry when accessing elements residing in foreign repositories. Finally, the integration framework now assigns the keys of the source and target repository to the link repository at runtime.

The registry provides the following operations:

[3]The registry currently supports the meta types *class* and *association*.

- "get(): RepositoryRegistry": Gets the singleton instance of this class which is unique in every Java Virtual Machine. Returns the one and only instance of this class.
- "registerRepository(String repositoryKey, RefPackage outermostPackage): boolean": Registers an outermost package of a repository at this registry. An outermost package may be registered with multiple unique keys. Each unique key identifies exactly one outermost package. Returns a boolean value specifying whether the outermost package is now registered with the given key.
- "unregisterRepository(String repositoryKey): RefPackage": Unregisters a formerly registered outermost package by removing the key from this registry. Returns the outermost package that was registered with the given key or null if no outermost package was registered with the given key.
- "unregisterRepository(RefPackage outermostPackage): RefPackage": Unregisters a formerly registered outermost package by removing all associated keys from this registry. Returns null if no keys are associated with the given outermost package or the outermost package otherwise.
- "getRepositoryKeys(): Set<String>": Gets all keys that are currently registered.
- "getRepositoryKeys(RefPackage outermostPackage): Set<String>": Gets all keys associated with the given outermost package.
- "getMetamodel(String key): RefPackage": Gets an outermost package that has been formerly registered with the given key. Returns the outermost package if it has been registered with the given key. Null otherwise.
- "getClass(String repositoryKey, String qName): RefClass": Looks up a class in the given outermost package. "qName" is the full qualified name of the class to be looked up in the outermost package. Returns the class if it exists in the outermost package. Null otherwise.
- "getInstance(String repositoryKey, String qName): RefObject": Convenience method that creates an instance of a class in the given outermost package. Returns an instance of the class with the specified name if it exists in the outermost package. Null otherwise.
- "getAssociation(String repositoryKey, String qName): RefAssociation": Looks up an association in the given outermost package. "qName" is the full qualified name of the class to be looked up in the outermost package. Returns the association if it exists in the outermost package. Null otherwise.

Furthermore, the registry can be switched with attribute *queryAllRepositories* into a mode that looks up all registered repositories when a query for the existence of a class or association is requested instead of looking up just the repository with the specified key. This especially increases robustness and takes effect if due to some issue a repository is reregistered with another key at the repository during runtime. However, this mode significantly slows down the execution time of operations "getClass" and "getAssociation".

8. Related Work

Based on the characterization of "useful" TGGs in Chap. 4 we proposed extensions to triple graph grammars in Chap. 5 and a new translation algorithm in Chap. 6 which is applicable by bidirectional language translators. In this chapter we compare our approach of mapping two different formal languages onto each other with several related approaches. Therefore, we have chosen a number of representatives of this sort of model transformation approaches. We start with a discussion of decision criteria for unidirectional and bidirectional model transformation approaches in Sect. 8.1. In Sect. 8.2 we will have a look at selected model transformation approaches. Section 8.3 gives an overview of the different approaches to triple graph grammars that are all based on TGGs as defined by Schürr in [Sch95]. The chapter is concluded by a summary of the discussed approaches in a feature matrix in Sect. 8.4.

8.1. Decision Criteria

In the following, we will review some of the criteria, published by Czarnecki et al. in [CH03] and [CH06], used to compare different model transformation approaches. Some of the criteria will be discussed in the text, whereas others will only be given in the feature matrix in Sect. 8.4.

As we have learned already in the preceding chapters, transformation rules consist of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS accesses the input model, whereas the RHS expands in the output model. Both LHS and RHS can be represented using any mixture of variables, patterns, and logic. Patterns can be string, term, and graph patterns. String patterns are used in textual templates, whereas model-to-model transformations (as in our case) usually use term or graph patterns. Patterns can be represented using abstract or concrete syntax of the corresponding input or output model language, and the syntax can be textual and/or graphical. Logic expresses computations and constraints on model elements. Logic may be non-executable or executable. We focus on transformation languages which provide an executable logic, which can take a declarative or imperative form. In the declarative form the program describes *what* should be accomplished. Whereas in the imperative form a control flow and an algorithm are given which describe *how* a problem should be solved.

Another aspect is *bidirectionality* of a rule which allows execution in both directions. In the case of TGGs, two so-called operational rules are derived from one bidirectional TGG rule, which allow to execute forward and backward transformations. Most other approaches are *unidirectional*. The effect is that if both forward and backward transformations are required these have to be specified in two separate rules which have to be kept in a consistent state manually.

We will also have a look at rule organization. Especially, we are interested in modularization, i.e. packaging rules into modules, and reuse mechanisms like inheritance between rules. In

addition we will have a look at the traceability support of transformation approaches. Some tools have integrated support for traceability, which allows, e.g., synchronization between models, i.e., incremental updates and determining the output of a transformation. Some approaches provide dedicated support for traceability, while others expect the user to encode traceability using the same mechanisms as for adding any other kinds of links in models. Some approaches with dedicated support for traceability require developers to manually encode the creation of traceability links in the transformation rules, while other create traceability links automatically.

Stevens discusses a framework for bidirectional transformations in [Ste10] based on thoughts about QVT. She focuses on basic requirements which bidirectional transformations should satisfy. The basic requirement that is pointed out by Stevens is *coherence*, i.e., satisfaction of the three conditions (1) *correctness* meaning that forward and backward transformations ensure that source and target model are in a consistent state according to the transformation relation, (2) *hippocraticness* or "check then enforce" semantics, i.e., transformations do not modify a pair of models if they are already in the specified relation, and (3) *undoability*, i.e., the ability to revert a modified model to the original version. Her theoretical approach is based on mathematical relations. She assumes that forward and backward transformations are expressible as mathematical functions. Furthermore, she states that a bidirectional transformation need not to be bijective. The behavior of a transformation may depend on the current value of the target model as well as on the source model. Furthermore, the behavior of a transformation should be deterministic, so that modeling it by a mathematical function is appropriate. The same transformation, given the same pair of models, should always return the same proposed modification. Her approach requires transformations to be total in the sense that forward and backward transformation are total functions.

The requirements for bidirectional transformations stated by Stevens are applicable to the implementation of bidirectional translators derived from a TGG specification. So, they become interesting when comparing different tool implementations of TGG-based translators. Instead we will focus on the comparison of different approaches to the TGG language in Sect. 8.3 and will not compare different tool implementations of TGGs.

8.2. Related Model Transformation and Model Integration Approaches

In this section we will discuss some selected model transformation approaches. Namely, we will have a look at ATL, Viatra2, Tefkat, ETL, AToM3, and GRoundTram. These approaches serve as a representative (but by no means complete) set of currently relevant model transformation approaches based on different technologies or formalisms. Most of them are unidirectional approaches and can, e.g., be used as underlying language for the definition of operational rules derived from a bidirectional TGG specification. Note that we have already discussed Story Driven Modeling (SDM), which is implemented in the CASE tool Fujaba and used by our tool MOFLON (cf. Chap. 7), in Sect. 2.8.3. In principle, all discussed model transformation approaches could be used to specify the operational rules derived from a TGG assumed that

they are feature compatible with SDMs. For a more comprehensive survey of bidirectional (model) transformation approaches we refer to [CFH+09]. As a representative of bidirectional model transformation approaches, we conclude this section with a discussion of QVT, the model transformation standard proposed by the OMG, and its relation to the TGG language.

8.2.1. ATL

ATL (ATLAS Transformation Language) is a hybrid model transformation language that allows both declarative and imperative constructs to be used in transformation definitions [JK05]. ATL is developed as part of the AMMA (ATLAS Model Management Architecture) platform. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models. A bidirectional transformation is implemented as a couple of transformations: one for each direction. In ATL rule inheritance can be used as a code reuse mechanism and also as a mechanism for specifying polymorphic rules. ATL is able to handle metamodels that have been specified according to either the MOF or the Ecore semantics [atl11a].

Declarative ATL rules are composed of a source pattern and of a target pattern. A source pattern specifies a set of source types and a guard (an OCL Boolean expression). A source pattern is evaluated to a set of matches in source models. The target pattern is composed of a set of elements. Every element specifies a target type and a set of bindings. A binding refers to a feature of the type (i.e. an attribute, a reference or an association end) and specifies an initialization expression for the feature value. Declarative rules are executed over matches of their source pattern. For a given match the target elements of the specified types are created in the target model and their features are initialized using the bindings. Executing a rule on a match additionally creates a traceability link in the internal structures of the transformation engine. This link relates three components: the rule, the match (i.e. source elements) and the newly created target elements.

ATL is accompanied by a set of tools that include the ATL transformation engine, the ATL integrated development environment (IDE) based on Eclipse [atl11b], and the ATL debugger. ATL transformations are compiled to programs in specialized byte-code. Byte-code is executed by the ATL virtual machine. The virtual machine is specialized in handling models and provides a set of instructions for model manipulation.

ATL and OMG's QVT (cf. Sect. 8.2.7) share some common features as they initially shared the same set of requirements defined in the QVT Request For Proposal—once ATL was one implementation of this proposal. However, actual ATL requirements have changed over time as this language matured [JK06].

8.2.2. Viatra2

Viatra [CHM+02] is a model transformation tool integrated into Eclipse and available under the Eclipse Public License [via11]. Viatra supports specification, design, execution, validation, and maintenance of transformations within and between various modeling languages and domains. Similarly to MOF and EMF it provides a proprietary model space for uniform representation of models and metamodels. Its transformation language has both declarative and

imperative features and is based upon formal mathematical techniques of graph transformation and abstract state machines [VB07]. The abstract state machines allow to assemble complex transformation programs. In Viatra, transformation rules are unidirectional. The transformation engine supports incremental model transformations and is realized as an interpretative approach. Contrary to traditional pattern matching approaches, Viatra's incremental pattern matching approach determines and updates matches in graphs after modification of the graph incrementally, which affects the efficiency of model-to-model transformations. When performing inter model transformations, a traceability model can be explicitly added and used separately. Consequently one could also use the Viatra framework for realizing a triple graph grammar engine.

8.2.3. Tefkat

Tefkat [LS05, tef11] is a model transformation engine which is embedded into the Eclipse Modeling Framework EMF. It realizes a declarative implementation of model transformation and is based on Frame Logic (F-logic) [KLW95]. The Tefkat language was designed specifically for the transformation of MOF models using patterns and rules. One of the design issues of Tefkat was to address the OMG's QVT request for proposal, which resulted in a language that is quite similar to the QVT Relations language. Change propagation, i.e., incremental transformations, can be supported through a model-merge process. In order to enable traceability, Tefkat uses so-called *tracking classes* to represent mapping relationships between source and target elements. A Tefkat transformation rule is unidirectional. It matches and then constrains objects either from the source model or from the trackings, and then creates a number of target model objects with a set of constraints. Besides a standalone implementation, Tefkat is also implemented as Eclipse plugin which provides a syntax-highlighting editor and supports debugging.

8.2.4. Epsilon Transformation Language

The Epsilon Transformation Language (ETL) [KRP11, KPP08] is a hybrid, rule-based model-to-model transformation language that is built upon the Epsilon Object Language (EOL) [KPP06]. Both are part of the Epsilon framework which is implemented as Eclipse GMT component. ETL is seamlessly integrated with a number of other task-specific languages to help to realize composite model management workflows, e.g., model-to-text transformation, model comparison, validation, merging and unit testing. The idea behind EOL is to provide the meta-modeling community with a general approach to access and manipulate models regardless of their underlying formalism (e.g., MOF, Ecore, KM3, etc.). EOL is an OCL-based imperative language that provides features such as model modification, multiple model access, conventional programming constructs (variables, loops, branches etc.), user interaction, profiling, and support for transactions. EOL was designed to overcome the restrictions of OCL but still comply to precise semantics. ETL is a hybrid approach allowing for declarative patterns and an imperative fallback to call EOL methods and ETL rules. Traceability is implicitly maintained that allows for additional cross model checking with EVL—the Epsilon Validation Language. The model transformation engine runs in batch mode and also in

a (semi-)interactive mode by specifying rules that prompt the user for input during their execution. Incremental model transformations utilize the previously established trace model and in addition regard specification information that defines which elements have to be kept. ETL is designed as a unidirectional transformation language. It is highly reusable due to its precise semantics of inheritance [WKK+11] and modules (via EOL modules). Furthermore, ETL is not limited to transformation specifications that cope with one source and target model only, but with m source and n target models simultaneously, i.e., an arbitrary number of source models can be transformed into an arbitrary number of target models.

8.2.5. AToM3

AToM3 is a meta-modeling tool which uses graph grammars as underlying mechanism for model transformations [dLV02]. Models are internally represented as Abstract Syntax Graphs (ASGs). The left- and right-hand sides of a graph pattern are modeled separately. Patterns in graph transformation rules may be enriched with additional textual conditions specified in Python or in OCL. The execution order of rules during graph rewriting is based on a user-defined priority.

AToM3 has been extended to support triple graph grammars based on type graphs with inheritance [GdL04, TEG+05]. A formalization of inheritance on type graphs with triple graph grammars was out of scope in our contribution. However, inheritance is supported in our implementation of TGGs in MOFLON. (Negative) application conditions are also supported in TGG productions based on [GdL04]. Moreover, AToM3 allows to model TGG productions which delete elements. However, AToM3 is not able to derive forward and backward translators from a TGG specification—which is done in our approach. Instead, a *triple graph replacement system* is generated from the triple specification. A user is now able to build models with this system according to the specified syntax. The system triggers events if the user performs a creation, editing, or deletion operation in one domain. This invokes the application of rules which modify the graphs of the opposite and correspondence domain. In [GdL04] this is called a combination of *event-driven grammars* and triple grammars. The effect is that modified elements in one domain allow for small incremental updates according to the invoked operation. The main advantage is that the triple graph replacement system which serves as an integration system always knows—thanks to the triggered events—which TGG production is applied by the user. Instead, our integration system, i.e., TGG algorithm, has to figure out which TGG production is applied on his own by parsing the input graph and comparing the situation in the input graph with patterns contained in the TGG productions. So, the approach to TGGs at translator level in AToM3 is orthogonal to our approach and, therefore, does not need to be evaluated in detail further here.

8.2.6. GRoundTram

GRoundTram (Graph Roundtrip Transformation for Models) is a system that can be used to build a bidirectional transformation between two models (graphs) [HHI+10, HHI+11]. A model transformation is described in UnQL+, which is functional (rather than rule-based) and compositional with high modularity for reuse and maintenance. UnQL+ is a high-level,

SQL-like programming language for describing queries / graph transformations in a textual syntax. In the GRoundTram system, all UnQL+ programs are first compiled (also called *desugared*) into UnCAL and then run. UnCAL (Unstructured Calculus) is the core graph algebra of UnQL+. It is rather low-level and more machine-friendly. UnCAL consists of a set of constructors for building graphs and a powerful structural recursion for manipulating graphs. This graph algebra has clear bidirectional semantics and is evaluated in a bidirectional manner.

Graphs in UnQL+ and UnCAL are rooted and directed cyclic graphs with no order between outgoing edges. They are edge-labeled in the sense that all information is stored as labels on edges and the labels on nodes serve as a unique identifier and have no particular meaning. Input and output graphs for an UnQL+ program are written in the UnCAL format. UnCAL can also describe graph transformations, not just graphs themselves. Metamodels are represented in the KM3 format [JB06] (Kernel MetaMetaModel—an implementation-independent language to write metamodels; developed at INRIA and supported under the Eclipse platform; structurally close to eMOF 2.0 and Ecore).

Due to the bidirectional semantics of UnCAL, a backward model transformation can be automatically derived from a forward model transformation, so that both transformations can form a consistent bidirectional model transformation. GRoundTram implicitly keeps trace information between source and target models after transformations which is then used for bidirectional transformations. The produced target model is a *traceable view* that has trace IDs assigned to all of its nodes. GRoundTram has been designed to support bidirectional transformations in the sense that after applying a forward transformation the system can also propagate changes on the target model back to the source (backward updating/evaluation). In database specific terms, a forward transformation is used to produce a target view from a source, while the backward transformation is used to reflect modification on the view to the source. So, GRoundTram interprets UnQL+/UnCAL source files for modified output graphs and produces an input graph in which modification on the output graph is reflected. Consequently, a backward transformation is performed after a forward transformation has been successfully executed. This is contrary to our approach which derives both forward and backward transformation rules from one single bidirectional rule which are executable independently from each other.

8.2.7. QVT

The OMG has standardized the transformation language QVT (Query / View / Transformation) in 2008 [Obj08]. QVT is based on MOF and OCL. QVT is mainly used to specify a transformation of models of one domain into models of another domain. Its architecture consists of a declarative and an imperative part. The declarative part consists of the two languages *Relations* and *Core*, which are on different abstraction levels. The QVT standard defines a mapping from the Relations language to the Core language. QVT Core is equally powerful to the Relations language, but QVT Relations is more user-friendly. The imperative part is made up of the *Operational Mappings* language and *Black Box* implementations. Only the declarative part is of relevance when comparing QVT with TGGs. QVT Relations supports complex object pattern matching and object template creation which is quite similar

to graph pattern matching. Correspondences of two (or more) domains are specified in a relational way. Besides a textual abstract syntax the Relations language supports a graphical representation[1], too. QVT Core is a small language that only supports pattern matching over a flat set of variables. Both QVT and TGGs establish traceability links while performing a transformation. QVT Relations implicitly defines the trace (meta)model, whereas it is explicitly defined by QVT Core and TGGs. QVT has a "check then enforce" semantics, which means that a transformation is only enforced if a former check reveals that the source and target models are in an inconsistent state.

Kurtev discusses the state of the art of QVT in [Kur08]. He gives an overview to the QVT languages and summarizes the currently available QVT tools.

Königs [Kön09] as well as Greenyer and Kindler [Gre06, GK10] investigated the similarities of QVT and TGGs. They found that both languages have much in common and that QVT can be realized by TGGs. Greenyer and Kindler propose a transformation of QVT Core mappings to TGG productions. This way QVT Core mappings can be translated into a TGG. In conjunction with the transformation specification from QVT Relations to QVT Core given in the QVT specification, both declarative languages of QVT may be executed by TGG engines. More generally, QVT can be realized with graph grammar-based approaches [LLVC06, RN08].

QVT Relational circumvents the efficiency versus completeness tradeoff problem as follows: it simply applies all matches of all translation rules to a given input model in parallel and merges afterwards elements of the generated output model based on key attributes. This approach is rather error-prone and requires a deep insight of the QVT tool developer as well as its users how rules match and interact with each other. As a consequence, [GK10] shows that today existing QVT Relational tools may produce rather different results when processing the same input. [GK10] identifies a semantic gap in the QVT standard which leads to different behavior in different QVT implementations. They discuss that QVT implementations use different interpretations of binding semantics. The QVT specification does not specify whether bindings of model patterns of different rule applications may overlap in the instance model. The result of a comparison of different QVT implementations in the master's thesis of Guan [Gua09] substantiates this statement. He found that different QVT implementations produce different results when transforming the same models according to a given transformation description. Contrary to this semantic gap in the QVT specification, TGGs use a precisely defined interpretation for variable bindings, which is called *bind-exactly-once semantics* in [GK10]. That is, every element of a completely translated input graph is bound to exactly one produced node of an application of a TGG production. In other words, every element is created by exactly one application of a TGG production. Consequently, every element of the input graph is translated by exactly one match of a translation rule. If there are elements that are not translated /bound in the end, then the translation process is incomplete in the TGG approach, i.e., not successful. The QVT specification states in this case that elements which are not bound in the end shall be deleted.

In QVT, the direction of a transformation, i.e., forward or backward, is chosen at runtime. *A transformation invoked for enforcement is executed in a particular direction by selecting one*

[1] The hexagonal element that represents a TGG link was inspired by the graphical representation of a relationship in the QVT Relations language.

of the candidate models as the target. [...] The execution of the transformation proceeds by first checking whether the relations hold, and for relations for which the check fails, attempting to make the relations hold by creating, deleting, or modifying only the target model, thus enforcing the relationship [Obj08, 7.1.1]. In the TGG approach, forward and backward translators are derived from a TGG specification.

Likewise to TGGs, QVT bears bidirectional capabilities. Whether a QVT specification is bidirectional depends on the specification itself. The QVT standard does not explicitly discuss this issue and only gives the following hint on bidirectionality: *Bi-directional transformations are only possible if an inverse operational implementation is provided separately* [Obj08, 6.3]. Another hint whether a specification is bidirectional is given by the keywords *checkonly* and *enforce* which are explicitly encoded in a QVT relation and are of relevance in conjunction with the transformation direction. *Whether or not the relationship may be enforced is determined by the target domain, which may be marked as checkonly or enforced. When a transformation is enforced in the direction of a checkonly domain, it is simply checked to see if there exists a valid match in the relevant model that satisfies the relationship. When a transformation executes in the direction of the model of an enforced domain, if checking fails, the target model is modified so as to satisfy the relationship, i.e., a check-before-enforce semantics* [Obj08, 7.2.3]. The examples given in the QVT standard never use a combination of *enforce/enforce* which is necessary in order to result in a really bidirectional transformation specification. Instead the examples use the combinations *checkonly/enforce* and *checkonly/checkonly*. When translating forward, the *checkonly/enforce* combination of domains A and B (domain A is checkonly, domain B is enforced) will adjust the target model (i.e., model of domain B) such that the relationship is satisfied. But, when translating backward the target model (now the model of domain A) is only checked but not adjusted. Consequently, one only specifies a unidirectional translator and a consistency checker when not using the combination *enforce/enforce* but the combination *checkonly/enforce*.

Both QVT and TGGs normally operate in three domains: domain A (or source domain), domain B (or target domain), and trace domain (or link or correspondence domain). However, neither QVT nor TGGs are in principle limited to three domains and a specification may contain a (theoretically) unlimited number of domains [KS05].

QVT as a transformation standard of the OMG does not come with a native implementation provided by the OMG. However, different implementations exist for the QVT languages. An overview of the currently active available QVT tools, which have been investigated in the master's thesis of Guan [Gua09], that provide support for QVT Relations and QVT Core is given subsequently.

medini QVT is an implementation of the QVT Relations language by ikv++ [ikv11]. Medini QVT features the execution of QVT transformations expressed in the textual concrete syntax of the Relations language. Medini QVT is based on Eclipse and provides an editor with code assistant and the possibility of debugging. Traces are managed which enables incremental update during re-transformation. The QVT key concept is supported which enables incremental update as well as the transition from manual modeling to automation through transformations in the absence of traces. In addition, bidirectional transformations are supported.

ModelMorf is a commercial tool from Tata Consultancy Services which supports the specification and execution of model transformations in the QVT Relations language [Tat11]. Modelmorf is a non open source implementation which also supports incremental execution. It currently does not come with an integrated development environment. Transformation rules have to be specified with an external editor and are then passed as an XML representation, e.g., in Ecore/EMF format, to the transformation engine.

M2M QVTR Model to model is an Eclipse project that aims at the implementation of the OMG QVT standard [Ecl11]. The M2M project will deliver a framework for model-to-model transformation languages. There are three transformation engines that are developed in the scope of this project: ATL, procedural QVT (Operational), and declarative QVT (Core and Relational). The QVT Declarative (QVTd) component aims to provide a complete Eclipse based IDE for the Core (QVTc) and Relations (QVTr) Languages defined by the OMG QVT Relations (QVTR) language. QVT Declarative currently provides editors, parsers, and meta-models for QVTc and QVTr. QVT Declarative will provide a dedicated perspective, an execution environment for QVTc and QVTo, and an integrated debugger for QVTc and QVTo.

8.3. Triple Graph Grammar Approaches

There are certain TGG approaches which are all based on the original TGG approach [Sch94, Sch95] from 1994. Some of them formally or informally extend or modify the original approach. In the following sections we will have a look at certain representative TGG approaches and compare them with our TGG approach.

The first TGG publication [Sch94] introduced a rather straight-forward construction of translators. It relied on the existence of graph grammar parsing algorithms with exponential worst-case space and time complexity. As a consequence a first generation of follow-up publications [Lef94, JSZ96] all made the assumption that the regarded graphs have a dominant tree structure and that the components of a TGG production possess one and only one primary node. Based on these assumptions an algorithm is used that simply traverses the tree skeleton of an input graph node by node and selects an arbitrary matching FGT/BGT rule for a regarded node that has a node of this type as its primary node. This algorithm defines translation functions that are neither correct nor complete in the general case. Both properties are endangered by the fact that the selected tree traversal order does not guarantee that rules are applied in the appropriate order. It may happen that the application of a rule fails because one of its context nodes has not been processed yet or that a rule is applied despite of the fact that one of its context nodes has not been matched by another rule beforehand.

NACs where introduced in [HHT96] in the context of model transformation approaches based on graph transformation. Most TGG approaches developed in the first 15 years refrain from the usage of NACs. Some of them even argued that NACs cannot be added to TGGs without destroying their fundamental properties! But, rather recently some application-oriented TGG publications simply introduced NACs without explaining how derived translation rules and their rule application strategies have to be adapted precisely. The publications

even give the reader the impression that NACs can be evaluated faithfully on a given input graph without regarding the derivation history of this graph with respect to its related TGG. [KW07], e.g., explicitly makes the proposal to handle complex graph constraints in this way, whereas [GGL05] ignores the problems associated with the usage of NACs completely. [GdL04, TEG+05] introduce NACs at TGG level but do not consider their handling at translator level (cf. Sect. 8.2.5).

8.3.1. Own Approaches

In [KKS07] we have discussed modularization concepts for TGG productions. The package concepts *nesting*, *import*, and *merge* as well as *inheritance on link types* have been presented. These concepts were out-of-scope in this thesis and, therefore, have not been discussed here.

In [SK08] we discussed translators based on TGGs with NACs that already guarantee correctness but not completeness for derived translators without introducing a backtracking algorithm, i.e., without trading efficiency for completeness. In [KLKS10] we presented efficient, correct, and complete translators based on TGGs with a certain subset of NACs, but without giving support for secondary elements and many-to-many relationships in source and target domain. Translating many-to-many relationships contained in graphs given to a TGG translator as input graph either require a combination of the patterns depicted in Figs. 5.11 (a) AND (b) in TGG productions OR the pattern depicted in Fig. 5.11 (c) in order to successfully parse cyclic or acyclic input graphs. Both possibilities are not supported by the algorithm presented in [KLKS10]: a combination of patterns (a) and (b) would result in a CycleInRecursiveContextTranslation error (cf. Sect. 6.6) and primary links as depicted in pattern (c) are not supported in TGG productions of this approach. Consequently, the algorithm presented in this thesis also does not support a combination of patterns (a) and (b)—due to the CycleInRecursiveContextTranslation error. But, primary links in TGG productions are supported in the algorithm presented in this thesis (cf. Sect. 4.3 and Fig. 5.11 (c) in Sect. 5.3.3) and so are many-to-many relationships in instance graphs that are translated by the algorithm.

In [KLKS10] we have introduced the so-called *dangling edge condition*. In this thesis, we lifted this rather basic condition which applies to the world of graphs to the world of models. That is, this condition checks for dangling links that are contained in a graph and are somehow represented as a combination of nodes and/or edges (cf. mapping of models to graphs in Sect. 2.8.1). Therefore, we have added a level of indirection to the dangling condition for TGGs.

8.3.2. Becker et al.

Becker et al. concentrate on consistency maintenance in incremental and interactive integration tools [BHLW07]. They especially focus on scenarios where user interaction is inherently required because the effects of changes cannot be determined automatically and deterministically. Therefore, they support user interactions so the user can control the integration process. Their approach is based on triple graph grammars. The underlying formalism uses directed, typed, and attributed graphs. Instead of story driven modeling its predecessor language PROGRES is used as specification language for graph transformations and also for

TGG productions. PROGRES does not use a compact notation for graph productions like SDM. Instead it uses the more detailed representation where left-hand and right-hand side are shown separately.

The approach distinguishes between primary and secondary elements attached to a TGG link in a TGG production. They are called dominant increment and normal increment respectively. Moreover, a TGG link is connected to context nodes of the source and target domain as well as to other TGG (sub)links.

Becker et al. present an algorithm which maintains consistency in integration tools. Likewise to our algorithm, the algorithm in [BHLW07] looks for conflicts among rules. But contrary to our approach it presents conflicts to the user, who performs a selection among the conflicting rules. The algorithm is specified using the PROGRES development environment [Sch91]. [BHLW07] decouples pattern matching from graph transformation, i.e., splits pattern matching and execution of FGT and BGT rules derived from TGG productions in order to allow conflict detection and user interaction during execution of the algorithm. This can be compared to our introduction of *core rules* which are used in our algorithm to find applicable rules. Unfortunately, Becker et al. do not give proofs for the properties *correctness*, *completeness*, and *efficiency* of their algorithm.

8.3.3. Giese and Wagner

Giese and Wagner focus on the efficient execution of transformation rules derived from a TGG and how to achieve an incremental model transformation for synchronization purposes [GW09]. They show that due to the speedup for the incremental processing in the average case even larger models can be tackled. Their approach is implemented as an extension of the TGG implementation of Fujaba. In the approach of [GW09], TGG links from the link domain are passed to forward and backward graph translation rules as parameter in the transformation algorithm. So, this approach attacks the rule ordering problem in a rather different way. It introduces a kind of controlled TGGs, where each rule explicitly creates a number of child rule instances that must be processed afterwards. Thus, one of the main advantages of a rule-based approach is in danger that basic rules can be added and removed independently of each other and that it is not necessary to encode a proper graph traversal algorithm explicitly. This is contrary to our proposed algorithm which passes elements from the input domain, i.e., source or target domain, to FGT /BGT rules. [GW09] does not discuss whether their algorithm satisfies the properties *correctness*, *completeness*, and *efficiency*. [HEO+11] tries to close this gap by discussing the properties correctness and completeness for an incremental algorithm that is based on the work on incremental synchronization by Giese and Wagner and further assumes that forward and backward propagation operations are deterministic.

8.3.4. Königs

In his PhD thesis [Kön09], Königs extends the original TGG approach of Schürr [Sch94, Sch95]. The resulting approach is based on recent (OMG) standards and integrates with the QVT standard where reasonable by means of syntax and concepts. Like in our approach the source

and target metamodels are specified as MOF-based metamodels. A TGG specification is translated into operational SDM rules from which executable code can be generated. The work of Königs directly precedes the work done in this thesis. The tool support that implements the concepts of [Kön09] which is provided by MOFLON has been reused and extended in this thesis. Königs does not precisely define his TGG approach by means of formal definitions like we do. Instead he specifies the metamodel of his approach by utilizing CMOF and relies on the formal foundations provided by the original TGG approach. Contrary to the original approach his approach bases on typed and attributed graphs. Our approach formally extends the original TGG approach and relies on the metamodel for TGGs specified in [Kön09].

Königs introduces TGG parameters, which are used as attribute values in a TGG production. By adding parameters to the declaration of TGG productions he clarifies the concept of specifying attribute value expressions and their processing at rule derivation time. This has been discussed also in this thesis. The following extensions of the TGG approach were made by Königs from which the implementation in MOFLON and, therefore, our approach indirectly benefits. He adopts MOF's concepts for modularization and reusability, which allows for modularization of TGG productions and using inheritance of link types. Moreover, TGG link types can be enriched with OCL constraints, which allows to constrain the correspondence domain. This was out-of-scope in our approach and therefore has not been formally introduced.

Königs introduces an algorithm which is used by TGG translators implemented in the integration framework. He adopts the concept for explicitly controlling the rule application order from QVT. The main ideas of this algorithm are: (a) traversing model elements in a top-down fashion, i.e., elements that serve as a container are traversed first. (b) The application of a rule is delayed until required (context) elements have been processed. (c) The algorithm applies all possible rules if multiple rules are applicable and (d) a rule is applied to all matches if multiple matches exist. Case (b) is contrary to our approach where context elements are eagerly translated whenever required. Moreover, our algorithm aborts its execution in case (c) and only applies a rule to at most one match if case (d) occurs. As a consequence, [Kön09] introduces an algorithm that still relies on a tree traversal, but keeps track of the set of already processed nodes and uses a waiting queue to delay the application of rules if needed. This algorithm defines correct translators, but has an exponential worst-case behavior concerning the number of re-applications of delayed rule instances.

[Kön09] intentionally does not support negative application conditions because they cause problems at rule derivation time. He argues that one does not know how to handle NACs when deriving operational translation rules. Instead the concept of rule priorities is supported, which is able to often simulate NACs. Rule application conflicts are resolved or avoided by using priorities because rules with a higher priority are chosen to be applied before a rule with a lower priority would be applied. Our approach supports a certain class of NACs and introduces the dangling edge condition which is evaluated at rule application time which allows to remove NACs from the input domain of translation rules. If needed our algorithm can be extended such that it also evaluates priorities of TGG productions. However, it is necessary to ensure that the fundamental properties correctness, completeness, and efficiency are not violated due to this extension.

8.3.5. Greenyer and Kindler

Greenyer and Kindler reconcile TGGs with QVT [GK10]. Their approach is described in detail in the master thesis of Greenyer [Gre06]. The TGG implementation of Greenyer is based on an interpretative approach to TGGs instead of compiling the TGGs. Their approach is an extension of the original TGG approach and is based on attributed and typed graphs. The implementation supports Multi Graph Grammars [KS05] which is a major improvement of TGGs because transformations can be specified between multiple models. They present a metamodel for TGGs aligned with the QVT metamodel and Ecore. A TGG editor has been implemented for the Eclipse platform using the Eclipse Graphical Modeling Framework (GMF). This editor is based on EMF and Ecore and allows to conveniently specify TGG productions and aids to design only such graph patterns which are syntactically correct according to the referenced domain models.

Greenyer and Kindler increase the expressiveness of their TGG approach by certain concepts. A construct that is equivalent to the herein discussed TGG parameters is supported. A so-called *attribute equality constraint* is used to specify that two attributes of different domains should have the same value. In addition, so-called *gray nodes* or *reusable nodes* are supported. These nodes are context nodes of a TGG production which match nodes in a host graph that do not need to be translated already by a previous call to a TGG translation rule. Similar to the original TGG approach, [GK10] allow TGG productions with more than one TGG link that is created. This is contrary to the approach specified in this thesis that explicitly allows only one TGG link that is created per TGG production in order to determine the primary elements in both domains. This way derived translators can distinguish primary from secondary elements in every domain. However, this is only a technical limitation of the approach in this thesis and may be resolved in another way such that also many TGG links which are created are supported in TGG productions. Negative application conditions are not supported by [GK10] but a related concept that allows to model so-called *NULL nodes*.

The algorithm of [Gre06] implemented in the interpreter simply applies TGG translation rules (if possible) as long as there are untranslated elements in the input graph. The transformation terminates either if there are no more input elements left to translate or if there are no more rules that can be applied. Multiple matches and multiple applicable rules are not handled by this algorithm. It is up to the specificator of the TGG to ensure that non-determinism is avoided and a confluent set of TGG productions are modeled.

8.3.6. Ehrig et al.

Ehrig et al. have produced most of the publications concerning the formalization of the TGG language. They have examined the fundamental properties of TGGs, i.e., *expressiveness*, *correctness*, *completeness*, and *efficiency*, in great detail from a formal point of view. Unfortunately, they do not present algorithms in their publications. Instead, they provide concepts which could be used as a formal foundation for an implementation of an algorithm.

Ehrig et al. started the formal construction and analysis of model transformations based on TGGs defined by Schürr [Sch94] and Königs and Schürr [KS06] in [EEE+07] by analyzing information preservation of bidirectional model transformations. There, a formal result is

presented that shows under which conditions a given forward transformation sequence has an inverse backward sequence in the sense that both together are *information preserving* concerning the source graphs. [EEE+07] extends the concept of triple graphs based on simple graphs to triple graphs based on typed, attributed graphs and on concepts from category theory.

Two formalisms ensuring that elements are translated only once, which is also a vital part of our translation algorithm, are discussed in [EP08] and [HEOG10, HEGO10]. [EP08] introduces so-called *kernel NACs* that ensure that a translation rule cannot be applied twice at the same match. [HEOG10, HEGO10] introduce so-called *translation attributes*, which keep track of the elements which have been translated already. Translation attributes are the formalized equivalent of the visual representation of the translation status as marked/unmarked checkboxes placed next to an element as used, e.g., here or in [KLKS10].

[EEE+07] has a main focus on correctness, whereas efficiency, expressiveness, and completeness are out-of-scope. Follow-up publications (e.g., [EHS09] and [EEHP09]) then introduced NACs defined in the source or target domain in an appropriate way and proved that translators may be derived from a TGG with NACs that are compatible with their TGG. Moreover, [EEHP09] discusses a formalism that constructs correct and complete model transformation sequences on-the-fly, i.e., correctness and completeness properties of a model transformation need not to be analyzed after completion, but are ensured by construction. This behavior is similar to the behavior of our algorithm which either produces a correct and complete translation or aborts in case of an error. Unfortunately, both [EHS09] and [EEHP09] trade efficiency for completeness. That is, neither [EHS09] nor [EEHP09] present an algorithm that is able to find an appropriate sequence of translation rules in polynomial time which is necessary to create efficiently working translators.

Therefore, [HEGO10] introduces so called *filter NACs* which are used in a translation process to avoid translation sequences that produce wrong output graph triples. [HEGO10] suggests either to use a static approach to generate filter NACs or a (semi-automatic) dynamic generation of filter NACs based on critical pair analysis. In the static approach a procedure is discussed that generates filter NACs which have the same purpose as our *dangling edge condition*: to avoid translation sequences that produce untranslated edges that are not translatable later on. In [HEGO10], filter NACs are embedded as usual NACs in a translation rule and are checked before the translation rule is applied. In our approach, each untranslated incident edge of a primary node is checked to satisfy the dangling edge condition for each translation rule candidate in a translation step executed by our algorithm. Our algorithm ensures that the dangling edge condition is satisfied before it applies a translation rule candidate. Both approaches avoid backtracking during a forward or backward translation which dramatically increases the efficiency of the translation process.

8.4. Comparison Matrix

In order to give a brief summary of the discussed model transformation approaches, the table depicted in Fig. 8.1 summarizes the features supported by the discussed model transformation approaches. A part of the comparison criteria has been taken from the requirements that

have to be met by languages that are used to specify bidirectional translators (cf. Sect. 3.5). Other criteria for comparison have been taken from the classification of model transformation approaches discussed in [CH03] and [CH06].

	ATL	Viatra2	Tefkat	ETL	AToM3	GRoundTram	QVT	SDM	TGG
general approach (bxL1)	+	+	+	+	+	+	+	+	+
precise semantics (bxL2)	+	+	+	+	+	+	o	+	+
formalism	-	graph transformation & abstract state machines	F-logic	epsilon object language	graph grammars	graph-based	-	graph grammars	graph grammars
expressiveness (bxL3)	+	+	+	+	+	+	+	+	o
type of transformation rules	unidirectional	unidirectional	unidirectional	unidirectional	both	bidirectional*	uni/bidirectional	unidirectional	bidirectional
reusability (bxL5)	rule inheritance	+	+	inheritance	o	o	+	o	o
modularization	-	+	-	+	+	+	o	-	+
tool support	+	+	+	+	+	+	+	+	+
declarative / imperative	both	both	declarative	both	declarative	declarative	both	both	declarative
syntax (graphical, textual)	textual	textual	textual	textual	graphical	textual	both	graphical	graphical
incrementality (bxT6)	-	+	-	+	+	+	+	-	+
traceability	+	-	-	+	+ (with TGGs)	+	+	-	+

Figure 8.1.: Feature comparison of discussed model transformation approaches.

9. Conclusion and Future Work

In this chapter we conclude this thesis by summarizing the main aspects discussed therein in Sect. 9.1. Finally, Sect. 9.2 closes this thesis by discussing some issues that are still subject for further investigation.

9.1. Conclusion

In Chap. 2, we discussed languages, model-driven engineering, models and graphs, and model and graph transformation. Especially the more precise term *language model* has been introduced for a particular subset of models that are called *metamodel* in the model-driven world. Moreover, we presented a mapping from models to graphs that includes the mapping of classes, associations, and properties to typed and labeled graphs. This mapping is used in the following chapters to demonstrate *how* graphs can be used as formal basis for model-driven purposes.

Chapter 3 then discusses the integration of languages in general utilizing a natural language translation analogy. After this introduction to the field of language translation we discussed the integration of formal languages in particular utilizing the well-known example of mapping class diagrams and relational database schemata. Therefore, the language models consisting of syntax and constraint definitions of class diagrams and database schemata have been discussed. Afterwards, productions have been presented that are used to construct valid model instances of class diagrams and database schemata according to the language specification. Then, a concrete mapping of class diagrams and database schemata is discussed informally. The chapter closes with a discussion of similarities in natural and formal language translation. There, the natural language translation example from the beginning of the chapter is related to the formal language translation example of class diagrams and database schemata. We found out that both approaches have many things in common: the need for automated language translators, usage of traceability links on a fine-grained level, involving language experts, loss of information, encoding of information, the need for bidirectional translators, and synchronization of information after changes. Summarizing we state challenges for realizing bidirectional translators. These challenges are related to the languages that are used to specify bidirectional translators as well as to the runtime level of bidirectional translators. Finally, the chapter presents our approach of model-driven language integration. This approach named *TiE* (tool integration environment) consists of techniques from the fields of requirement engineering, metamodeling, model transformation, code generation, and tool adaptation. The resulting products generated by TiE are operated by our tool integration framework.

The mapping discussion of class diagrams and database schemata is used in Chap. 4 to specify a bidirectional formal mapping with triple graph grammars (TGGs). Triple graph

grammars are a formalism that is able to specify the bidirectional simultaneous evolution of two related languages in a declarative and visual style. The main advantage of triple graph grammars is the derivation of forward and backward translators from a set of bidirectional TGG productions. The formalism behind TGGs is reviewed and the set of TGG productions for integrating class diagrams and database schemata is built. Finally, the fundamental properties of TGGs and derived translators is discussed. The main aspects are the expressiveness of the specification language and the termination, correctness, completeness, and efficiency of derived language translators.

Chapter 5 now discusses extensions of (simple) triple graph grammars. Lifting of triple graph grammars based on simple graphs to extended triple graph grammars based on typed and attributed graphs is discussed. The first main result of this thesis is the introduction of a well-defined class of negative application conditions for restricting the applicability of TGG productions. Therefore, so-called *integrity-preserving productions* are introduced which must fulfill the property that an application of such a production does not destroy the integrity of a graph, i.e., the graph must satisfy a set of constraints after production application. The second main result of this thesis is given in Theorem 1 in Sect. 5.2.2. This theorem directly affects the derivation of forward and backward translation rules from a TGG production which are utilized by forward and backward translators. The consequence of this theorem is that NACs can be safely removed from the input component of derived translation rules if the translation algorithm checks the integrity of generated graphs with respect to their sets of constraints. The third main result of this thesis is the introduction of the *local completeness criterion* for triple graph grammars. This criterion is used to prove the completeness property of translators derived from a TGG specification. As the fourth main result of this thesis, the so-called *dangling edge condition* is introduced to the world of triple graph grammars. This condition mainly states that after translating a node of the input graph every incident edge that is not currently translated must be translatable in future translation steps. The dangling edge condition is regarded by our translation algorithm introduced in the next chapter.

Now, Chap. 6 discusses the translation level of TGGs, i.e., translators that utilize rules derived from a TGG specification. At first, a framework is presented that can be used by translation algorithms based on triple graph grammars. So-called *core rules* are discussed which are used by algorithms to identify context elements of a given node that is about to be translated. Moreover, these core rules are used to handle secondary elements during the translation process. Then, as an introductory step, a simple graph translation algorithm is discussed. The main disadvantage of this algorithm is that it does not fulfill the completeness property of TGG-based translators. Consequently, as the fifth main result of this thesis, a more advanced graph translation algorithm is introduced. This algorithm is based on the results for extended triple graph grammars as discussed in Chap. 5 that are still expressive enough to cope with common mapping situations. [Sch94] and other TGG related publications have shown *that* it is possible to build efficient and compatible translators. But they did not show *how* to build them. We have shown, with the advanced algorithm in Listing 4, *how* to build efficient translators that are compatible with their TGG. The presented algorithm can be utilized by bidirectional working translators derived from certain TGG specifications. This algorithm is rather efficient and correct as well as complete regarding to the TGG corresponding to the derived translators. The algorithm takes triples of graphs as input but it treats these graphs

as models and especially treats links as nodes. This is possible due to the ability of mapping models to graphs as presented by a mapping of MDA-style models into the semantic domain of typed graphs in the category of graphs and total graph morphisms (cf. Sect. 2.8.1). The TGG productions supported by this algorithm are either based on graph patterns that represent model transformation patterns—consisting of higher-level constructs: object, link, slot, value, and classifier—or graph patterns—consisting of lower-level constructs: node, edge, and graph morphism. After discussing the algorithm, its properties *termination*, *efficiency*, *correctness*, and *completeness* are proved. The chapter is completed by a discussion of a simple consistency check algorithm. This algorithm is able to check for a given set of TGG links, whether these are consistent with their TGG specification.

Chapter 7 presents the implementation of the approach of this thesis. The approach is implemented in the meta-CASE tool MOFLON. MOFLON is based on the CASE tool Fujaba and supports story driven modeling (SDM), OCL, code generation, and triple graph grammars. The generated code for a TGG specification can be executed in our tool integration framework, which is able to visualize both source and target models as well as TGG links.

Finally, we discuss related work in Chap. 8. We see that other TGG approaches have also extended the original approach of triple graph grammars. But, the main difficulty is to be able to derive translators from an extended TGG specification that still fulfill the fundamental properties efficiency, correctness, and completeness. Especially efficiency and completeness compete against each other.

9.2. Future Work

In this thesis we identified a class of TGGs with NACs for which we can guarantee the completeness of the developed algorithm that are useful in practice due to a polynomial runtime complexity. But, we limited NACs to only prevent creating invalid graphs according to the graph constraints given in the source and target graph languages. This limited class of TGGs with NACs has to be enlarged but compatibility and efficiency properties of derived translators have to be ensured. Definition 15 in Sect. 5.2.1 limits productions such that invalid graphs never become valid after applying a production where NACs are removed. How other classes of NACs are to be treated in translators is still an open question. We suppose these NACs to be equivalent to partial/temporal constraints. Such NACs have another character because an output graph triple tends to temporarily violate corresponding constraints. Therefore, one cannot replace the check for a NAC with a check for its corresponding constraint at any time like we did for the class of NACs supported in this thesis.

The introduced *dangling edge condition* is only able to examine edges directly incident to a node, i.e., the look-ahead used for examination is 1. It is up to future work to increase this look-ahead by a meaningful amount, so also indirectly referenced nodes are taken into account.

It is also up to future work to develop decision criteria which can be checked already at compile time and which guarantee that the translation rules derived from a given TGG never raise errors at runtime. For example, the presented translation algorithm stops with an error message whenever it encounters a situation where more than one rule is applicable to translate

a specific input graph element. Confluence checking techniques should offer the right means for the detection and classification of potential rule application conflicts at compile time. The confluence checking algorithms of the tool AGG [HKT02], which are based on critical pair analysis, could be adapted for this purpose. We could also use results from Ehrig et al. (cf., e.g., [EEHP09]) regarding confluence so we are able to cope with situations where multiple rules are applicable that would lead to equivalent output graph triples. In this way we would be able to guarantee already at compile time that a graph translator derived from a specific class of TGGs will not stop its execution with an error instead of generating an existing output graph for a given input graph. Furthermore, constraint verification techniques of the tool GROOVE [Ren08] should allow to check the here introduced requirements already at compile time: (a) TGG productions never create graph triples that violate graph constraints of the related schema, (b) NACs are only used to block graph modifications that would violate a graph constraint, (c) TGG productions never repair constraint violations by rewriting an invalid graph into a valid graph, and (d) TGGs fulfill the local completeness criterion. Until then, TGG developers have to design and test their TGGs carefully such that TGG productions do not violate the presented conditions of integrity-preserving productions.

To further increase expressiveness of the triple graph grammar formalism rule priorities could be added to the specification level of TGG productions (cf. Sects. 5.3.2 and 8.3.4). There are different possibilities to implement rule priorities in the presented algorithm (cf. Listing 4 in Sect. 6.6). The algorithm can be modified such that the different activities of the algorithm operate on ordered sets of rules and the activities are only executed for the rules with the highest priority. Rules with a higher priority would be checked first and rules with lower priority are only taken into account if the algorithm fails for rules with higher priority. Another possibility is to modify the activity of the algorithm in Listing 4 that checks for competing core matches. The modification would only throw an according error if the competing matches where found by rules with the same priority. Consequently, line 46 of algorithm in Listing 4 would select one rule with the highest priority. Besides the modification of the algorithm, the correctness and completeness proofs have to be revisited, so it is guaranteed that these important properties are still fulfilled after the modifications for rule priorities.

An important issue for future work is to further study and improve techniques for incremental updates performed by translators, i.e., improve support for model synchronization [GW09]. In this thesis, model synchronization and incremental updates were out-of-scope, but support for this issue is urgently required in practice. Successfully performing incremental updates bears advantages in performance especially when dealing with large models. Moreover, additional information that is only present in one domain is not lost during retranslation. But, as history shows, incremental updates are difficult to realize in practice because models that were formerly consistent are modified and have to be synchronized later on. Due to these modification operations existing corresponding structures need to be deleted, cleaned up, and reconstructed. In order to determine which elements need to be deleted, dependencies between TGG links have to be considered. The challenge during a cleanup is to carefully delete information from source, target, and correspondence domain and to reuse as much of the existing structures as possible. Incremental updates only make sense if the cleanup process is less costly then a new translation or if additional information is kept during synchronization. It is up to future work to use the presented algorithm in Listing 4 for synchronization operations

or, in general, to guarantee the properties efficiency, correctness, and completeness also for incremental algorithms.

A first step towards incremental translations based on the presented algorithm has been made recently in [LK11]. There, *precedence triple graph grammars* are presented that are used for a topological sorting of the elements of the input graph given to a TGG-based translator. The sorted set of input elements aims at providing the elements created by the sequence of corresponding TGG productions in an appropriate order. The right sequence of translation rules is derived from this ordered set of elements. Another advantage of precedence TGGs is their ability to detect cycle errors, like discussed in Sect. 6.6, already at compile time. It is up to future work to integrate the results from precedence TGGs into the proposed framework for translation algorithms.

Concepts for modularization and inheritance of TGGs have been discussed in [KKS07] and [WKK+11] but are still an issue for future work. The results for related TGG productions, e.g., creation of TGG links that inherit from the same TGG link type, have to be further investigated. It is likely that such TGG productions are similarly assembled which demands for concepts that avoid replication of elements which form a TGG production at specification time. A formal introduction of inheritance concepts to node and TGG link types of the underlying graph formalism for TGGs is discussed in [LK11]. Inheritance of node types in the input graph affects the creation of *Legal Node Creation Context* (LNCC) tuples (discussed in Sect. 5.3.2). The synthesis of LNCC tuples from TGG productions has to be reconsidered if inheritance is introduced on node types, which results in additional tuples that also consider inherited node types.

Evaluations of the here presented class of TGGs with NACs in research cooperations with our industrial partners, where TGGs are used to ensure consistency of design artifacts, will show whether our claim is true that the here introduced new class of TGGs is still expressive enough for the specification of a sufficiently large class of bidirectional model translations that are needed in practice.

Bibliography

[AKK+08] Carsten Amelunxen, Felix Klar, Alexander Königs, Tobias Rötschke, and Andy Schürr. Metamodel-based Tool Integration with MOFLON. In *30th International Conference on Software Engineering*, pages 807–810, Leipzig, Germany, May 10 - 18 2008. ACM New York, NY, USA. Formal Research Demonstration.

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In A. Rensink and J. Warmer, editors, *ECMDA - Foundations and Applications*, volume 4066 of *LNCS*, pages 361–375. Springer Verlag, 2006.

[Amb03] Scott W. Ambler. *Agile Database Techniques.* John Wiley & Sons, 2003.

[Ame09] Carsten Amelunxen. *Metamodel-based Design Rule Checking and Enforcement.* PhD thesis, Technische Universität Darmstadt, 2009.

[atl11a] ATL - EclipseWiki. www, August 2011. `http://wiki.eclipse.org/ATL`.

[atl11b] ATL at Eclipse. www, August 2011. `http://eclipse.org/atl/`.

[Bé04] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE, The European Journal for the Informatics Professional*, V(2):21–24, April 2004.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, pages 273–280, Washington, DC, USA, 2001. IEEE Computer Society.

[BHLW07] Simon M. Becker, Sebastian Herold, Sebastian Lohmann, and Bernhard Westfechtel. A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. *Software and Systems Modeling (SoSyM)*, 6(3):287–315, September 2007.

[Bic04] Lutz Bichler. *Codegeneratoren für MOF-basierte Modellierungssprachen.* PhD thesis, Universität der Bundeswehr München, 2004.

[Bru06] Jean-Michel Bruel, editor. *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *LNCS*. Springer, 2006.

[Cam09] Matthew Campbell. A Graph Grammar Methodology for Generative Systems. Technical report, University of Texas at Austin, Department of Mechanical Engineering, 2009. published in UT Faculty/Researcher Works collection.

[CEM+06] Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors. *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *LNCS*. Springer, 2006.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations: Second International Conference, ICMT2009*, volume 32 of *Lecture Notes in Computer Science (LNCS)*, pages 260–283, Heidelberg, June 2009. Springer Verlag.

[CH11] Hugh Chisholm and Franklin Henry Hooper, editors. *The Encyclopeadia Britannica*. Encyclopaedia Britannica, New York, 11th edition, 1911.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CHM+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, page 267, Washington, DC, USA, 2002. IEEE Computer Society.

[dCLF93] Dennis de Champeaux, Douglas Lea, and Penelope Faure. Domain Analysis. In *Object-Oriented System Development*, chapter 13. Addison Wesley, 1993.

[dLV02] Juan de Lara and Hans Vangheluwe. AToM3: A Tool for Multi-Formalism and Meta-Modelling. In *FASE'02*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.

[Ecl11] Eclipse. Model To Model (M2M). www, July 2011. `http://www.eclipse.org/m2m/`.

[EEE+07] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabi Taentzer. Information Preserving Bidirectional Model Transformations. In *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*. Springer, 2007.

[EEHP09] Hartmut Ehrig, Claudia Ermel, Frank Hermann, and Ulrike Prange. On-the-Fly Construction, Correctness and Completeness of Model Transformations based on Triple Graph Grammars. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems. Proceedings of MODELS 2009*, volume 5795 of *LNCS*, pages 241–255. Springer, 2009.

[EEKR97] Ehrig, Engels, Kreowski, and Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific Publishing, 1997.

[EEKR99] Ehrig, Engels, Kreowski, and Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific Publishing, 1999.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Series. Springer-Verlag, 2006.

[Ehr79] Hartmut Ehrig. Introduction to the Algebraic Theory of Graph Grammars (A Survey). In *Proceedings of the International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes In Computer Science (LNCS)*, pages 1–69, London, UK, 1979. Springer-Verlag.

[EHRT08] Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors. *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008, Proceedings*, volume 5214 of *LNCS*. Springer, 2008.

[EHS09] Hartmut Ehrig, Frank Hermann, and Christoph Sartorius. Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. *ECEASST*, 18, 2009.

[EP08] Hartmut Ehrig and Ulrike Prange. Formal Analysis of Model Transformations Based on Triple Graph Rules with Kernels. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabi Taentzer, editors, *Proceedings of the International Conference on Graph Transformation (ICGT'08)*, volume 5214 of *LNCS*, pages 178–193, Heidelberg, 2008. Springer.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph Grammars: an Algebraic Approach. In *IEEE Conf. on Automata and Switching Theory*, pages 167–10, 1973.

[ERRS10] Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors. *Graph Transformations, 5th International Conference, ICGT 2010, Twente, The Netherlands, September 27–October 2, 2010, Proceedings*, volume 6372 of *LNCS*. Springer, 2010.

[FMRS07] Christian Fuss, Christof Mosler, Ulrike Ranger, and Erhard Schultchen. The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES. In *Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 6 of *Electronic Communications of the EASST*, 2007.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 157–167, 2000.

[FP10] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. Addison Wesley, 2010.

[GdL04] Esther Guerra and Juan de Lara. Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation. In *Graph Transformations, Second International Conference, ICGT 2004*, volume 3256 of *LNCS*, pages 54–69. Springer, 2004.

[GGL05] Lars Grunske, Leif Geiger, and Michael Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In A. Hartman and D. Kreische, editors, *First European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA*, volume 3748 of *LNCS*, pages 284–298, Nuremberg, Germany, November 2005. Springer.

[GGZ+05] Lars Grunske, Leif Geiger, Albert Zündorf, Niels van Eetvelde, Pieter van Gorp, and Daniel Varro. Using Graph Transformation for Practical Model-Driven Software Engineering. In *Model-Driven Software Development*, pages 91–117. Springer Berlin Heidelberg, 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GK07] Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In *Model Driven Engineering Languages and Systems (MoDELS)*, volume 4735 of *LNCS*, pages 16–30, Nashville, TN, USA, 2007.

[GK10] Joel Greenyer and Ekkart Kindler. Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. *Software and Systems Modeling*, 9(1):21–46, 2010. Special Section Paper.

[Gre06] Joel Greenyer. A Study of Model Transformation Technologies: Reconciling TGGs with QVT. Diploma thesis, Universität Paderborn, July 2006.

[Gua09] Yuan Guan. Vergleich von QVT-Implementierungen mit Triple Graph Grammars. Master thesis, Technische Universität Darmstadt, July 2009. (in German).

[GW09] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8(1):21–43, 2009.

[HC07] Reiko Heckel and Alexey Cherchago. Structural and behavioural compatibility of graphical service specifications. *J. Log. Algebr. Program.*, 70(1):15–33, 2007.

[HEGO10] Frank Hermann, Hartmut Ehrig, Ulrike Golas, and Fernando Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In J. Bézivin, R.M. Soley, and A. Vallecillo, editors, *Proc. Int. Workshop on Model Driven Interoperability (MDI'10)*, MDI '10, pages 22–31, New York, NY, USA, 2010. ACM.

[HEO+11] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In *14th International Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, 2011. (accepted for publication).

[HEOG10] Frank Hermann, Hartmut Ehrig, Fernando Orejas, and Ulrike Golas. Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Proceedings of the International Conference on Graph Transformation (ICGT' 10)*, volume 6372 of *LNCS*, pages 155–170. Springer, 2010.

[Hes06] Wolfgang Hesse. More matters on (meta-)modelling: remarks on Thomas Kühne's "matters". *Software and Systems Modeling*, 5(4):387–394, 2006.

[HHI+10] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing Graph Transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 205–216, New York, NY, USA, 2010. ACM.

[HHI+11] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. *GRoundTram Version 0.9.2 User Manual*, May 2011. `http://www.biglab.org/pdf/manual.pdf`.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.

[HJS71] Hans Jürgen Schneider. Chomsky-like systems for partially ordered symbol sets. Technical Report Informationsverarbeitung II, Technische Universität, Berlin, 1971.

[HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *Proceedings of the First International Conference on Graph Transformation*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.

[Hof99] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Penguin Books, 1999. 20th-anniversary Edition (the original appeared 1979).

[HR00] David Harel and Bernhard Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff. Technical Report MCS00-16, Weizmann Science Press of Israel, 2000.

[ikv11] ikv++ technologies ag. medini QVT. www, July 2011. `http://projects.ikv.de/qvt/`.

[JB06] Frédéric Jouault and Jean Bézivin. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *LNCS*, pages 171–185, Bologna, Italy, 2006.

[JBR99] Ivar Jacobsen, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JK05] Frederic Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138, Montego Bay, Jamaica, October 2-7 2005.

[JK06] Frederic Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC'06)*, pages 1188–1195. ACM New York, NY, USA, 2006.

[JSR02] *Java™ Metadata Interface (JMI) Specification, Version 1.0*, June 2002. JSR 040 Java Community Process; Specification Lead: Ravi Dirckze, Unisys Corporation.

[JSZ96] Jens Jahnke, Wilhelm Schäfer, and Albert Zündorf. A Design Environment for Migrating Relational to Object Oriented Database Systems. In *12th International Conference on Software Maintenance (ICSM'96)*, pages 163–170, 1996.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In *The 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 285–294, Dubrovnik, Croatia, September 03 - 07 2007. ACM New York, NY, USA.

[KLKS10] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel, editors, *Graph Transformations and Model-Driven Engineering*, volume 5765 of *LNCS*, pages 141–174. Springer, 2010.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, 1995.

[Kön05] Alexander Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS 2005*, Montego Bay, Jamaica, 2005.

[Kön09] Alexander Königs. *Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation*. PhD thesis, TU Darmstadt, January 2009.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Object Language (EOL). In *Proceedings of the European Conference in Model Driven Architecture (EC-MDA)*, pages 128–142. Springer, 2006.

[KPP08] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin / Heidelberg, 2008.

[KRP11] Dimitrios Kolovos, Louis Rose, and Richard Paige. *The Epsilon Book*. unpublished, 2011. `http://www.eclipse.org/gmt/epsilon/doc/book/`.

[KRS08] Felix Klar, Sebastian Rose, and Andy Schürr. A Meta-Model-Driven Tool Integration Development Process. In *2nd International United Information Systems Conference*, Lecture Notes in Business Information Processing (LNBIP), pages 201–212. Springer, April 2008.

[KRS09] Felix Klar, Sebastian Rose, and Andy Schürr. TiE - A Tool Integration Environment. In *Proceedings of the 5th ECMDA Traceability Workshop*, volume WP09-09 of *CTIT Workshop Proceedings*, pages 39–48, 2009. `http://www.utwente.nl/projecten/ecmda2009/workshops/ECMDA2009-TW.pdf`.

[KS05] Alexander Königs and Andy Schürr. MDI - a Rule-Based Multi-Document and Tool Integration Approach. *Special Section on Model-based Tool Integration in Journal of Software&System Modeling*, 2005. accepted for publication.

[KS06] Alexander Königs and Andy Schürr. Tool Integration with Triple Graph Grammars - A Survey. *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques, Electronic Notes in Theoretical Computer Science*, 148:113–150, 2006.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling*. John Wiley & Sons, 2008.

[Küh06a] Thomas Kühne. Clarifying matters of (meta-) modeling: an author's reply. *Software and Systems Modeling*, 5(4):395–401, 2006.

[Küh06b] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.

[Kur08] Ivan Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 5088 of *Lecture Notes in Computer Science*, pages 377–393, 2008. Third International Symposium, AGTIVE 2007, Revised Selected and Invited Papers.

[KW07] Ekkart Kindler and Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, Department of Computer Science, University of Paderborn, Germany, June 2007.

[KWB03] Kleppe, Warmer, and Bast. *MDA Explained.* Addison-Wesley, 2003.

[Lef94] Martin Lefering. Software Document Integration Using Graph Grammar Specifications. In *6th International Conference on Computing and Information*, volume 1 of *Journal of Computing and Information*, pages 1222–1243, 1994.

[LK11] Marius Lauder and Felix Klar. Precedence Triple Graph Grammars. In *Proceedings of the International Symposium of Applications of Graph Transformation With Industrial Relevance (AGTIVE 2011)*, 2011. (accepted for publication).

[LLVC06] Laszlo Lengyel, Tihamer Levendovszky, Tamas Vajk, and Hassan Charaf. Realizing QVT with Graph Rewriting-Based Model Transformation. In *Proceedings of the Second International Workshop on Graph and Model Transformation (GraMoT)*, volume 4 of *Electronic Communications of the EASST*, Brighton, UK, September 2006.

[LS05] Michael Lawley and Jim Steel. Practical Declarative Model Transformation with Tefkat. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 139–150, Montego Bay, Jamaica, October 2-7 2005.

[MCG05] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A Taxonomy of Model Transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[Met05] Andreas Metzger. A Systematic Look at Model Transformations. In *Model-Driven Software Development*, pages 19–33. Springer Berlin Heidelberg, 2005.

[MFB09] Pierre-Alain Muller, Frederic Fondement, and Benoit Baudry. Modeling Modeling. In *Model Driven Engineering Languages and Systems*, volume 5795 of *LNCS*, pages 2–16. Springer, 2009.

[MKU04] Stephen J. Mellor, Scott Kendall, and Axel Uhl. *MDA Distilled.* Addison-Wesley Professional, 2004.

[Nei80] James M. Neighbors. *Software Construction Using Components.* PhD thesis, Department of Information and Computer Science University of California, Irvine, 1980.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 742–745, New York, NY, USA, 2000. ACM.

[Obj06] Object Management Group. Meta Object Facility (MOF) Core Specification, January 2006. formal/06-01-01.

[Obj07] Object Management Group. MOF 2.0/XMI Mapping, Version 2.1.1, December 2007. formal/2007-12-01.

[Obj08] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v1.0, April 2008. formal/2008-04-03.

[Obj09a] Object Management Group. Unified Modeling Language Infrastructure Version 2.2, February 2009. formal/2009-02-04.

[Obj09b] Object Management Group. Unified Modeling Language Superstructure Version 2.2, February 2009. formal/2009-02-02.

[Obj10a] Object Management Group. MOF Facility Object Lifecycle (MOFFOL), March 2010. formal/2010-03-04.

[Obj10b] Object Management Group. Object Constraint Language, February 2010. formal/2010-02-01.

[PD90] Rubén Prieto-Díaz. Domain Analysis: An Introduction. *SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

[Per08] Stephen J. Perrault, editor. *Merriam-Webster's Advanced Learner's English Dictionary.* Merriam-Webster, Incorporated, 2008.

[PR69] John L. Pfaltz and Azriel Rosenfeld. Web Grammars. In D. E. Walker and L. M. Norton, editors, *Proceedings of the 1st International Joint Conference On Artificial Intelligence*, pages 609–620, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[Qui37] Willard Van Orman Quine. Logic Based on Inclusion and Abstraction. *The Journal of Symbolic Logic*, 2(4):145–152, December 1937.

[Rao84] Jean-Claude Raoult. On Graph Rewriting. *Theoretical Computer Science*, 32:1–24, 1984.

[RE67] Hugo Riemann and Hans Heinrich Eggebrecht, editors. *Riemann Musiklexikon Sachteil.* Schott Music, Mainz, 1967.

[Ren08] Arend Rensink. Explicit State Model Checking for Graph Grammars. In R. De Nicola, P. Degano, and J. Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 114–132. Springer Verlag, Berlin, June 2008.

[RN08] Arend Rensink and Ronald Nederpel. Graph Transformation Semantics for a QVT Language. *Electronic Notes in Theoretical Computer Science*, 211:51–62, April 2008.

[RS97] Jan Rekers and Andy Schürr. Defining and Parsing Visual Languages with Layered Graph Grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[Sch91] Andreas Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. PhD thesis, TH Aachen, 1991.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. Technical Report AIB-94-12, RWTH Aachen, Fachgruppe Informatik Germany, 1994. (extended version).

[Sch95] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *20th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163, Heidelberg, 1995. Springer.

[SHE06] Catherine Soanes, Sara Hawker, and Julia Elliot, editors. *Oxford English Dictionary*. Oxford University Press, 6th edition, 2006.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation – the Heart and Soul of Model-Driven Software Development. *Software, IEEE*, 20(5):42–45, 2003.

[SK08] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In *4th International Conference on Graph Transformation*, volume 5214 of *Lecture Notes in Computer Science (LNCS)*, pages 411–425, Heidelberg, 2008. Springer Verlag.

[SNZ08] Andy Schürr, Manfred Nagl, and Albert Zündorf, editors. *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, volume 5088 of *LNCS*. Springer, 2008.

[Som07] Ian Sommerville. *Software Engineering*. Addision Wesley, 8th edition, 2007.

[Ste10] Perdita Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software and Systems Modeling (SoSyM)*, 9(1):7–20, January 2010.

[Tat11] Tata Consultancy Services. Modelmorf. www, July 2011. `http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`.

[tef11] Tefkat. www, July 2011. `http://www.tefkat.net`.

[TEG+05] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Daniel Varro, and Szilvia Varro-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *Model Transformation in Practice (MTiP'05), workshop at MODELS'05*, Jamaica, 2005.

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214 – 234, 2007. Special Issue on Model Transformation.

[via11] VIATRA2 Documentation. www, July 2011. `http://wiki.eclipse.org/VIATRA2`.

[WKK+11] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitrios Kolovos, Richard Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In *ICMT2011 - International Conference on Model Transformation*, Zurich, Suisse, June 2011.

[ZKB02] Jürgen Ziegler, Christoph Kunz, and Veit Botsch. Matrix browser: visualizing and exploring large networked information spaces. In *Proceedings of the CHI EA '02 extended abstracts on Human factors in computing systems*. ACM New York, NY, USA, 2002.

[Zün01] Albert Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation Thesis.

Index

A. Glossary

The most important symbols and terms used throughout this thesis have been collected and summarized in this chapter for the reader's convenience.

A.1. Terminology of Graphs

node Basic element. Represents (an instance of) something.

edge Basic element. An edge e *connects* two nodes n_1 and n_1. Edges used throughout this thesis are directed. An edge starts at its *source node* $s(e)$ and ends at its *target node* $t(e)$. The source node is also called *initial node* and the target node is called *head node*[1] in order to avoid collisions with the terms "source/target graph", "domain", and "component" when talking in TGG terms. Each of the two nodes has a specific role in the context of an edge. The role depends on whether a node is the initial node or the head node, i.e., either n_1 or n_2 may take the role of the initial node or head node. The subscript of a node (e.g., 1 or 2) does not automatically denote whether the node is the source or target of an edge.

adjacency Two nodes n_1 and n_2 are called *adjacent* if an edge exists between them.

neighborhood Nodes adjacent to a node n not including n itself are the *(open) neighborhood* of n and denoted $N(n)$. When n is also included, it is called a *closed neighborhood* and denoted $N[v]$.

incidence If an edge e connects two nodes n_1, n_2 these two nodes are said to be *incident* to that edge and the edge is said to be *incident* to those two nodes. The set of incident edges of a node n is denoted $inc(n)$.

elements Nodes and edges.

graph A set of elements.

graph (homo)morphism A mathematical construct to *relate* (the elements of) two graphs.

(graph) production A construct for creating (and modifying and deleting) elements of a host graph. Also *graph rewriting rule*. Consists of a set of elements. The elements of a production are contained in the *left-hand side* (LHS) L, the *right-hand side*(RHS) R, or the *set of negative application conditions* (NACs). Elements contained in the LHS

[1] "Head" because this node appears at the arrow head.

must be present in the host graph in order to apply the production. The RHS specifies, in conjunction with the LHS, the elements that are created ($R \backslash L$) and the existing elements that are deleted ($L \backslash R$) when applying the production. NACs specify which elements must not be present in the host graph in order to apply the production.

match A match *identifies* elements in a graph which are related to elements in a production. A match is a morphism from a production to a graph.

redex A redex is a subgraph within a host graph matching a given production's LHS or RHS. That is, a redex is a set of elements in the host graph identified by a match of a production in a host graph.

A.2. Terminology at TGG Level

graph triple A set of three (related) graphs. A triple of graphs consisting of a *source graph* G_S, a *target graph* G_T, and a *correspondence graph* G_C (also *link graph*). The correspondence graph relates (corresponding elements of) the source and target graph. Note that the initial and head node of an edge are contained in the same graph. The correspondence graph "links" to the source and target graph via morphisms.

(graph) production triple $p := (p_S \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$ Also: *TGG production.* A construct for simultaneously creating elements in the three graphs of a graph triple. That is, the production triple is used to evolve graphs of the graph triple simultaneously. Consists of three related production components (i.e., graph productions p_S, p_C, and p_T) that rewrite the source, correspondence, and target graphs.

triple graph grammar (TGG) A graph grammar based approach for specifying languages of graph triples. The core part of a TGG specification is a set of TGG productions. Source-to-target translators as well as target-to-source translators can be derived from a TGG specification. In order to simplify the development process of translators the production components of a TGG production are *monotonic* (i.e., they do not delete elements). Certain rules may be derived from a TGG production which are used to perform certain operations on a graph triple. For example, a production triple may be split into pairs of *operational rules* (r_S, r_{ST}) and (r_T, r_{TS})—a *local rule* and its corresponding *translation rule*. The elements *created* by a local rule are *translated* by its corresponding translation rule. These elements are the *to-be-translated elements* identified by a *core match*. Other rules derived from a single production triple are, e.g., used to check a graph triple for consistency or for restoring consistency after certain elements in some graph of a graph triple were changed.

source domain The elements contained in the source graph of a graph triple are said to be in the *source domain*.

target domain The elements contained in the target graph of a graph triple are said to be in the *target domain*.

link domain The elements contained in the correspondence graph of a graph triple are said to be in the *link domain*.

A.3. Terminology at Translator Level

The terms presented in the following two subsections enable to speak either in terms that depend on the direction of a translator (i.e, *forward* or *backward*) or in terms that are independent of the direction of a translator.

A.3.1. Translation Direction Dependent

forward graph translator (FGT) A translator based on the TGG language. An FGT *translates* a graph of the source domain to a corresponding graph of the target domain. A specific FGT which is based on a specific TGG (e.g., TGG_{CDDS}) translates, e.g., class diagrams to database schemata.

backward graph translator (BGT) A translator based on the TGG language. A BGT translates a graph of the target domain to a corresponding graph of the source domain. A specific BGT which is based on a specific TGG (e.g., TGG_{CDDS}) translates, e.g., database schemata to class diagrams.

source-local rule r_S An operational rule derived from a production triple. Used to create elements of the source domain only. Consists of three production components of which the link and target domain component are empty.

forward graph translation rule r_{ST} Also *source-to-target domain translating production triple.* An operational rule derived from a production triple. Used by a FGT in order to simulate the simultaneous evolution of a graph triple. That is, the translator produces the elements in link and target domain which correspond to the elements in source domain. This translation rule consists of three components of which the source component is read-only.

target-local rule r_T An operational rule derived from a production triple. Used to create elements of the target domain only. Consists of three components of which the source and link domain component are empty.

backward graph translation rule r_{TS} Also *target-to-source domain translating production triple.* An operational rule derived from a production triple. Used by a BGT in order to simulate the simultaneous evolution of a graph triple. That is, the translator produces the elements in source and link domain which correspond to the elements in target domain. This translation rule consists of three components of which the target component is read-only.

A.3.2. Translation Direction Independent

input domain The domain of a graph triple which a translator uses as main source of information to produce an output. Either the *source domain* (in the case of an FGT) or the *target domain* (in the case of a BGT).

output domain The domain of a graph triple in which a translator produces elements which correspond to the elements in the input domain. Either the *target domain* (in the case of an FGT) or the *source domain* (in the case of a BGT).

(graph) translator Translates a graph of the input domain to a corresponding graph of the output domain. Typically consists of a static part—a framework—(e.g., translation algorithms that do not depend on a specific TGG) and a dynamic part (e.g., a set of operational rules derived from a specific TGG).

GT_{in} The graph triple that is given as input to a translator.

GT_{out} The graph triple that is produced as output by a translator.

G_{input} The graph of the input domain. A component of a graph triple. Either G_S (in the case of an FGT) or G_T (in the case of a BGT). In common cases a translator does not modify the graph of the input domain during translation, i.e., G_{input} is read-only.

G_{output} The graph of the output domain. A component of a graph triple. Either G_T (in the case of an FGT) or G_S (in the case of a BGT).

translated elements TX The set of elements (nodes and edges) of the input domain that are already translated.

(input-)local rule r_I Either r_S (in the case of an FGT) or r_T (in the case of a BGT).

(input-to-output domain) translation rule r_{IO} Either r_{ST} (in the case of an FGT) or r_{TS} (in the case of a BGT).

m_I Match $(m_I, \varepsilon, \varepsilon)$ of a local rule's left-hand side in the input graph G_I identifies the context elements required by a local rule r_I.

m'_I Match $(m'_I, \varepsilon, \varepsilon)$ of a local rule's right-hand side in the input graph G_I identifies the context elements which were required by a local rule r_I and the elements that have been created by r_I. Also known as *core match* in the context of a translation rule r_{IO}.

m_{IO} Match of the full left-hand side $m_{IO} := (m'_I, m_C, m_O)$ of a translation rule r_{IO}.

$m'_I(L_I)$ Part of the match m'_I which identifies the elements in the input graph G_I that are required as context by a local rule r_I. These elements are also required as context by the corresponding translation rule r_{IO} (i.e., the context elements of a core match).

$m'_I(R_I \backslash L_I)$ Part of the match m'_I which identifies the elements that have been **created** by a local rule r_I. This part of the match also identifies the elements that will be **translated** by a translation rule r_{IO} (i.e., the to-be-translated elements of a core match).

A.4. Original TGG Terms Related With Terms Used in This Thesis

Table A.1 relates the terms used in the original TGG publication [Sch95] and its extended version [Sch94] with the terms used in this thesis.

[Sch94] / [Sch95]	this thesis	description
vertex	node	
$GT := (LG \leftarrow lr{-}CG{-}rr \rightarrow RG)$	$GT := (G_S \overset{h_S}{\leftarrow} G_C \overset{h_T}{\rightarrow} G_T)$	graph triple
$p := (lp \leftarrow lh - cp - rh \rightarrow rp)$	$p := (p_S \overset{h_S}{\leftarrow} p_C \overset{h_T}{\rightarrow} p_T)$	production triple p
$lp := (LL, LR)$	$p_S := (L_S, R_S)$	source component of p
$cp := (CL, CR)$	$p_C := (L_C, R_C)$	correspondence component of p
$rp := (RL, RR)$	$p_T := (L_T, R_T)$	target component of p
GT $\sim$p(g)$\sim>$ GT'	$GT \overset{p@m}{\rightsquigarrow} GT'$	graph triple rewriting
left to right translation	forward translation	
right to left translation	backward translation	
GT$\sim p_L \sim>$HT $\wedge$ HT$\sim p_{LR} \sim>$GT'	$GT \overset{r_S}{\rightsquigarrow} GT_S \overset{r_{ST}}{\rightsquigarrow} GT'$	simulated simultaneous evolution of GT (forward translation)
left-local production triple p_L	source-local rule r_S	
left-to-right translating production triple p_{LR}	forward graph translation rule r_{ST}	
lg'	m'_S	important morphism of LR-Splitting of Production Triples
expressiveness of TGG formalism	expressiveness of TGG formalism	a property of (bidirectional) translation languages
efficiently working graph parser	efficiency of translators	a property of translators
correctness of results produced by algorithm	correctness of translators wrt. a TGG	a property of translators
completeness of algorithm	completeness of translators wrt. a TGG	a property of translators
termination of algorithm	termination of translation process	a property of translators

Table A.1.: TGG terms used in [Sch94] and [Sch95] related with the terms used in this thesis.

B. Issues in Original TGG Publication

In the following we hint to some issues in the extended version [Sch94] of the original TGG publication [Sch95]. In conjunction with the terminology in Appendix A and especially Table A.1 the following hints will hopefully assist TGG novices while studying the original TGG publication.

Section 3 References to figures in Sect. 3 are broken in the text. All numbers referencing a figure in Sect. 3 have to be incremented by one to restore the consistency of these references.

Definition 3.1 (4) The definition of the morphism functions h_V and h_E contains two typos (cf., e.g., [EEPT06] for the definition of a graph morphism). The given definition
"$\forall e \in E : h_V(s(e)) = s(h_E(e)) \wedge h_V(t(e)) = t(h_E(e))$"
has to be adjusted to
"$\forall e \in E : h_V(s(e)) = s'(h_E(e)) \wedge h_V(t(e)) = t'(h_E(e))$"

Proposition 3.6 LR-Splitting of Production Triples Proposition 3.6 gives a formal introduction to LR-splitting of production triples p into p_L and p_{LR}. The proof of Proposition 3.6, which refers to Fig. 7, demands that all production applications (p, p_L, and p_{LR}) use the same morphism lg' (depicted in Fig. 7) to select an image of graph LR (i.e., right-hand side R_S of p) in LG' (i.e., source graph G_S that is rewritten by p_L/p_S and p into G'_S). The preceding paragraph to Proposition 3.6 gives an informal introduction to LR-splitting. It refers to morphisms lg that have to be kept fixed when applying first p_L and then p_{LR} (in order to simulate application of p). One has to be careful to not mix the morphisms lg from this informal introduction to Proposition 3.6 with the morphism lg depicted in Fig. 7. In general these morphisms lg are not identical!

Proposition 3.9 Computing LR-Translations This proposition sketches an algorithm that may be used to compute forward translations. The algorithm contains the following pseudo-code fragment involved in selecting a match (morphism lg) while translating the input graph, where $new_{LR}(p)$ are the to-be-translated elements in the input graph. "CoveredElements" are the elements in the source domain which have been translated so far and the corresponding elements in the link and target domain which have been created by translation rules:
"with $lg(new_{LR}(p)) \cap$ CoveredElements $\neq \emptyset$"
The intention of this fragment is to select the morphism lg if the to-be-translated elements are not yet translated. Therefore, the intersection of to-be-translated and covered elements must be the empty set. Consequently, the code fragment should be:
"with $lg(new_{LR}(p)) \cap$ CoveredElements $= \emptyset$".

C. Curriculum Vitae

09/2005 - 09/2011	Research assistant at *Technische Universität Darmstadt*
09/1999 - 08/2005	Student assistant at *Fraunhofer Institut für Graphische Datenverarbeitung (IGD)* in Darmstadt
10/1998 - 04/2005	Studies of Computer Science at *Technische Universität Darmstadt*
06/1997	Abitur at Gymnasium *Hohe Landesschule Hanau*

Erklärung laut §9 Promotionsordnung

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, den 15.09.2011 Felix Klar