TECHNISCHE
UNIVERSITÄT
DARMSTADT

Incremental Model Synchronization with Precedence-Driven Triple Graph Grammars

DURCH DEN FACHBEREICH 18
ELEKTRO- UND INFORMATIONSTECHNIK
ZUR ERLANGUNG DER WÜRDE EINES
DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

MARIUS PAUL LAUDER, M.SC.

GEBOREN AM

17. JUNI 1982 IN HALLE AN DER SAALE

REFERENT: PROF. DR. RER. NAT. A. SCHÜRR
KORREFERENT: PROF. DR. RER. NAT. H. GIESE

TAG DER EINREICHUNG: 28. NOVEMBER 2012
TAG DER MÜNDLICHEN PRÜFUNG: 11. FEBRUAR 2013

HOCHSCHULKENNZIFFER D17
DARMSTADT 2013

The work of Marius Lauder was supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt.

Please cite this document as:

URN: urn:nbn:de:tuda-tuprints-33520
URL: http://tuprints.ulb.tu-darmstadt.de/id/eprint/3352

This document was provided by tuprints,
TU Darmstadt E-Publishing-Service
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Marius Paul Lauder, M.Sc.: *Incremental Model Synchronization with Precedence-Driven Triple Graph Grammars*, © 28. November 2012

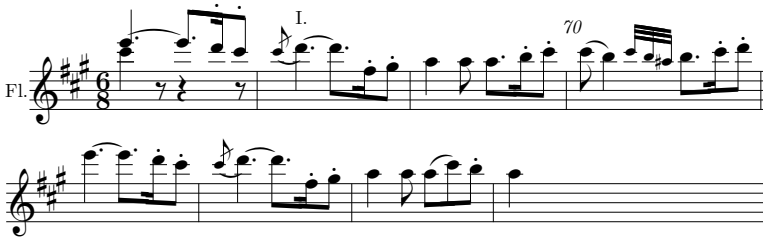
DECLARATION OF AUTHORSHIP

I warrant that the thesis presented here, is my original work and that I have not received outside assistance. All references and other sources used by me have been appropriately acknowledged in the work. I further declare that the work has not been submitted for the purpose of academic examination, either in its original or similar form, anywhere else.

I hereby grant the Real-Time Systems Lab the right to publish, reproduce and distribute my work.

Darmstadt, 28. November 2012

Marius Paul Lauder, M.Sc.



[Ulm09, p. 206]

DANKSAGUNG (ACKNOWLEDGMENTS)

Wie sich jeder vorstellen kann, ist eine Dissertation bei weitem keine Einzelleistung. Man wird von so vielen Seiten unterstützt, dass auf dem Deckblatt eigentlich noch viel mehr Namen stehen müssten.

Danken möchte ich zuallererst meiner Familie: Meine Eltern haben es mir mit viel Blut, Schweiß und Tränen ermöglicht diesen Weg einzuschlagen. Auch in schwierigen Situationen haben sie mich immer unterstützt und dabei selber auf so vieles verzichtet. Meine geliebte Frau und mein Sohn haben es geschafft, mich auch an tristen Tagen wieder aufzubauen und mich auf das Wesentliche zu besinnen. Jede Minute im Sandkasten und jedes herzhafte Kinderlachen haben mir oft ins Gedächtnis gerufen wofür sich all die Arbeit lohnt. Besonders bei diesen beiden muss ich mich für ihre Rücksicht und Ausdauer bedanken, so oft zurückzustecken. Noch herzlicher möchte ich mich für mein vieles Fehlen, sowohl körperlich als auch geistig, in den letzten Monaten entschuldigen. Nicht zu vergessen sind meine Großeltern und Münchner Tanten, die mir stets als Vorbilder des bewussten Handelns und Nachdenkens und als moralische Stützen zur Seite standen.

Inhaltlich wäre die Ausarbeitung ohne das Mitwirken von drei Personen im Besonderen undenkbar gewesen: Mein Doktorvater Andy Schürr und meine beiden Kollegen und Freunde Tony und Greg. In schier unermesslicher Geduld und Zeit haben diese drei sich mit mir die Köpfe darüber zerbrochen, wie man Definitionen und Beweise umsetzt. Scheinbar ausweglose Situationen zwei Tage vor der Paperdeadline wurden mit ihnen gemeistert und ins Gegenteil verkehrt. In unzähligen Diskussionen wurden Konzepte entwickelt, Beispiele durchgespielt und Whiteboards vollgeschrieben.

Danken muss ich außerdem ganz generell den Kollegen am Fachgebiet. Sie unterstützten mich mit Ideen, Anmerkungen oder einfach mehr Zeitpuffer z.B. durch das Übernehmen von Reviews in kritischen Situationen. Studenten haben in vielen Stunden und Tagen das programmiert, was ich mir gewünscht habe, oder die Konzepte untersucht, die zum runden Bild noch gefehlt haben. Nicht zuletzt möchte

ich Ingo danken, der mit tatkräftigem Support immer rechtzeitig zur Stelle war (danke an dieser Stelle auch an S.M.A.R.T. und seine gerade noch rechtzeitigen Warnungen) und das obwohl er das Klappern des Whiteboards so tapfer ertragen musste.

Prinzipiell muss ich auch der Landesbibliothek Wiesbaden und der Bibliotheksleitung der EBS danken, dass ich ihre Räumlichkeiten und, noch wichtiger, die Abgeschiedenheit nutzen konnte, um in stundenlanger Isolation konzentriert zu arbeiten. Zwar ohne Isolation, aber dafür auch ein Ort des fokussierten Schreibens war für mich stets der bequeme Sessel einer bekannten Kaffeehauskette.

Ihr alle habt diese Ausarbeitung erst möglich gemacht.
Danke für eure Unterstützung!

ABSTRACT

Triple Graph Grammars (TGGs) are a rule-based technique with a formal background for specifying bidirectional model transformation and, hence, can be applied to transform a given model into another and vice versa. In practice, models are either created from scratch by using a single input model, or incrementally synchronized by propagating changes between integrated models.

The outstanding property of incremental model synchronization is that in average only small portions of the whole model have to be retransformed as mostly only a subset of a model has been changed.

Hence, we have the opportunity to (i) improve efficiency of model transformations and (ii) to retain as much information as possible. Regarding information preserving capabilities, this offers the chance to qualitatively improve the results of model transformations. This is because additional model content (e.g., model elements or user specific decision during the actual transformation process), which is not covered by the model transformation itself, will be mostly retained.

In practical scenarios, unidirectional rules for incremental forward and incremental backward transformation are automatically derived from the specified TGG rules, and the overall transformation process is governed by a control algorithm. Current incremental approaches either have a runtime complexity that depends on the size of related models and not on the number of changes and their affected elements, or do not pursue formalization to give reliable predictions regarding the expected results, or impose such restrictions on the language of TGGs that the remaining expressiveness is not capable of certain real-world scenarios.

For these reasons, the aim of this thesis is to develop a novel approach to incremental model synchronization with TGGs that (i) is efficient regarding the number of changes, (ii) retains as much information as possible, (iii) complies with important formal properties, and (iv) is expressive enough for real-world scenarios.

Therefore, we introduce an incremental model synchronization algorithm for TGGs, which employs a static analysis on TGG specifications to efficiently determine the range of influence of model changes at runtime and, thus, to regard only these elements for synchronization.

Together with further improvements and critical discussions we will be able to show that this approach is a suitable means for complex model synchronization tasks.

Tripelgraphgrammatiken (TGGen) sind ein regelbasierter und formal fundierter Modelltransformationsansatz. Im praktischen Einsatz werden Modelle typischerweise entweder vollständig aus einem Eingabemodell abgeleitet oder Änderungen werden inkrementell in ein anderes Modell propagiert.

Die herausragende Eigenschaft der inkrementellen Modellsynchronisation ist, dass im Durchschnitt nur ein Teil des gesamten Modells bei der Transformation betrachtet werden muss.

Konsequenzen hieraus sind, dass zum einen Modelltransformationen effizienter werden und zweitens, dass so viele Informationen wie möglich erhalten bleiben. Der Erhalt von Informationen verbessert Modelltransformationen qualitativ, da zusätzlicher Modellinhalt, der nicht durch die angewandte Modelltransformation abgedeckt wird, erhalten bleiben kann. Dies können zum einen zusätzliche Modellelemente sein, oder zum anderen nutzerspezifische Entscheidungen während des Transformationsprozesses, die ohne Informationserhalt wieder manuell eingebracht werden müssen.

In der Praxis werden unidirektionale Regeln zur inkrementellen Vorwärts- bzw. Rückwärtspropagierung automatisch aus der TGG-Spezifikation abgeleitet und von einem sog. Kontrollalgorithmus verwendet. Aktuelle TGG-Ansätze haben teilweise schon eine Laufzeitkomplexität, die nur von der Anzahl der geänderten und deren abhängigen Elemente beeinflusst wird, sind aber nicht oder nur teilweise formalisiert, oder schränken die Ausdrucksmächtigkeit so ein, dass realistische Szenarios zum Teil nicht mehr abdeckbar sind.

Deshalb verfolgt diese Arbeit das Ziel einen neuen Ansatz für die inkrementelle Modellsynchronisation mit TGGen zu etablieren. Anforderungen an diesen Ansatz sind, dass er (i) effizient arbeitet (Komplexität abhängig von der Anzahl der Änderungen und deren Abhängigkeiten), (ii) so viele Informationen wie möglich während der Transformation erhält, (iii) wichtige formale Eigenschaften erfüllt und (iv) ausreichend Ausdrucksmächtigkeit besitzt, um auch für komplexe Szenarien anwendbar zu sein.

Es wird ein inkrementeller Kontrollalgorithmus präsentiert, der die Ergebnisse einer statischen Analyse von TGG-Spezifikationen nutzt, um den Einflussbereich von Modelländerungen zu erkennen und darauf aufbauend nur diese Elemente während der Synchronisierung zu betrachten. Mit weiteren Verbesserungsvorschlägen und kritischen Diskussionen wird sich zeigen, dass der hier vorgestellte Ansatz ein passendendes Mittel ist, um Modelle inkrementell zu synchronisieren.

CONTENTS

LIST OF FIGURES

Part I

INTRODUCTION, MOTIVATION & STATE OF THE ART

1 INTRODUCTION

Software engineering has never been an easy task [Wir08] since the dawn of scientific computation machines [FF03] and modern computers. Briefly, *software* is a generic description of how a system, regarding its physical possibilities should behave. Software, therefore, is an input to the machine and, it additionally defines how to process data and return an appropriate output. Developing software systems requires major knowledge about the desired functionality, runtime environment, customer expectations and many more. Software development [Bal00, LL06, Som07] is a hand-crafting discipline which started to evolve within the last century very rapidly and nowadays influences our daily life nearly everywhere.[1] Literally no daily task in our western world is independent from engineered software systems. As software affects nearly every aspect of our daily life, software cannot be developed without explicit rules of creating, testing and delivering. As [Bal00, LL06, Som07] describe, this should be true for all kinds of software.

Software Engineering

Software engineering has established itself as the hand-crafting art of building software. Hence, software engineering stands in one line with other engineering disciplines such as electrical or mechanical engineering. Software engineering covers the whole range of the software development process, starting from requirements engineering over coding guidelines and best practices to software maintenance.

A complex software systems is not built from different components only, but also from various libraries programmed in different languages by a number of software engineers. To cope with this high amount of interdependencies, a number of approaches has been developed which formulate best practices for software systems engineering. Typically, these approaches start with a more or less informal system description that will be used throughout the whole further engineering process as a reference for how the final result should look like and behave. As an example, consider the common V-Model XT [HH08, FHKS09, BdBfI10] which is often used in Germany to develop software systems, as shown in Fig. 1.

1 As a reference, interested readers are referred to the so-called Software Atlas [Lei10, LW11] and Software Monitor [KL10] provided by the Fraunhofer ISI, which monitor the production of software in Germany.

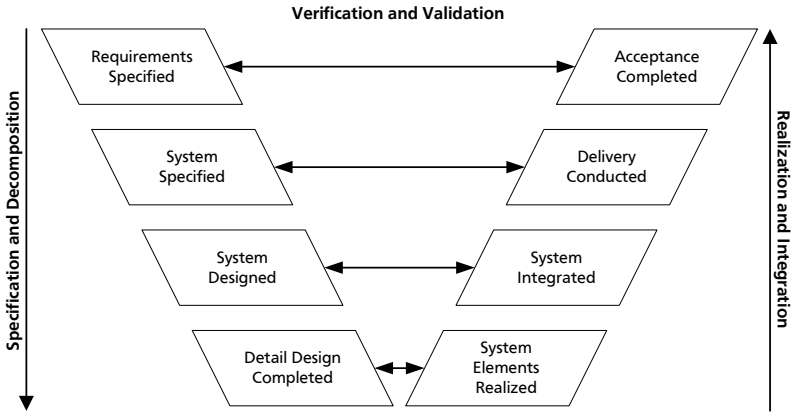


Figure 1: Overall process of the V-model XT [BdBfI10]

The process starts with developing so-called design documents such as requirements analysis and feature descriptions. Inherently, such descriptions are hard to understand as long as they are not formal and, therefore, allow for different interpretations. Consequently, there is a risk to develop incompatible software components based on the erroneous assumption that all engineers interpreted informal specifications in the same manner.

Model-Driven Software Development

To emphasize on such formal and concise specifications, *model-driven software development* (MDSD) has been developed where *models* (i.e., system specifications) play the central role throughout the overall software engineering process. The idea behind model-driven approaches is to stepwise refine and concretize structural and behavioral properties until the final product (e.g., a Java program) is created. Such architectural and behavioral data shall be retained in a so-called *model*. A model in this regard reflects existing facts of the real-world while abstracting from certain aspects and being utilized for a certain purpose.

MDSD is also referenced to as *Model-Driven Software Engineering* (MDSE) or *Model-Driven Architecture* (MDA). Based on common standards for expressing the design and architecture of a software system so-called *modeling languages* [ISO86, OMG11, OMG12] have been introduced, which provide means to design the structure and the behavior of a software, and how it interacts with users (and many more aspects). The probably best known representative of a modeling language is the *Unified Modeling Language*™ (UML) [OMG11] developed by the Object Management Group (OMG).

According to Stachowiak [Sta73] via [LL06], models are distinguished between being *descriptive* or *prescriptive*. The first describes some-

thing that can be found in the real-world (e.g., photographies of a building) while the latter describes something that has to be created according to the model (e.g., blueprints of a building). Hence, a model always (a) reflects an existing fact of the real-world, (b) abstracts from certain aspects, and (c) is used for a certain purpose. These properties are depicted in Fig. 2.

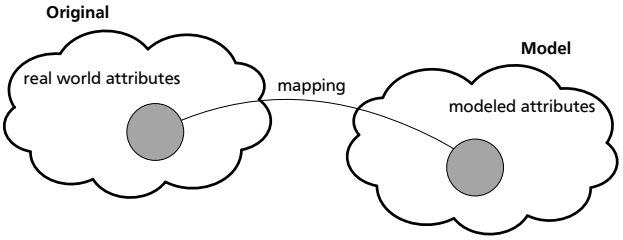


Figure 2: Model properties according to [Sta73] via [LL06]

Furthermore, Ludewig and Lichter [LL06] differentiate between models that describe artifacts produced in an engineering process and such models, which describe the engineering process itself. For the rest of this thesis, models in our sense refer to artifacts of engineering processes.

The model-driven software development process starts with specifying informal text documents, use cases and many more before formal models are created manually from this information. In a next step, these models are stepwise automatically refined (generated) to contain more platform specific information. Finally, it should be possible to generate from such platform specific models (PSM) program code that represents the desired product. The overall model-driven process is depicted in Fig. 3.

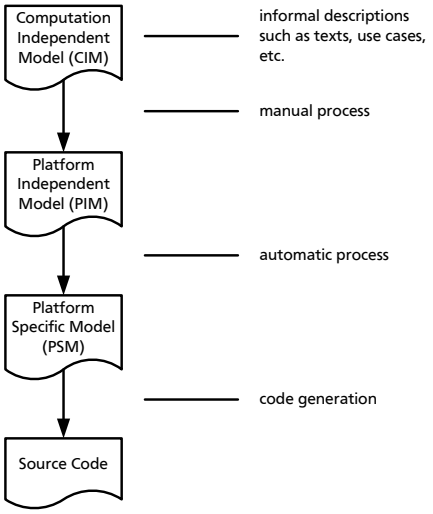


Figure 3: MDA process of building a software product according to [MM03]

Example

As an example, consider the development of an object-oriented Java program. At first, informal design specifications are created, which reflect the users' needs and expectations. This step coincides with the very first step of the V-Modell XT (cf. Fig. 1). As a further refinement step, UML class diagrams are developed, which reflect basic architectural aspects of the informal specification documents. Step by step, these design documents are enriched with additional information in the next phase of the V-Modell XT (e.g., aspects and interfaces of the to be used middleware framework) and, finally, actual program artifacts (i.e., Java code) are produced.

The set of legal models is described with a *metamodel* which defines the concepts (and their relationships) in a given domain. The prefix *meta* originates from Ancient Greek and stands for *beyond* or, according to the Oxford English Dictionary, for *about*. Hence, a metamodel describes a language in which models are legal instances and, therefore, *speaks about* models.

Example

Furthermore, it is possible to define a set of legal metamodels with a meta-metamodel and so forth. In the previously mentioned MDA, a four-layer metadata architecture has been introduced [MM03]. Figure 4 depicts the four layer architecture using terms from our Java development example. On the lowest layer M_0, called information layer, instances of real-world objects are placed. The depicted objects represent Java objects at runtime. These objects are instances of programmed classes of the next layer M_1, the so-called model layer. The metamodel layer M_2 defines the modeling language concepts such as Java classes and associations. Finally, the fourth layer M_3 describes generic concepts on a even higher level of abstraction.

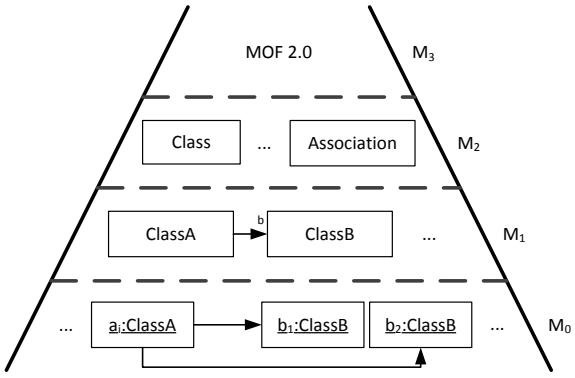


Figure 4: Exemplary four layer metadata architecture

Briefly, an element on layer M_{n-1} is an instance of an element on layer M_n. As a consequence a (meta-)model on layer M_n defines all models on layer M_{n-1} precisely. The single metamodel on layer M_3 is an instance of itself, thereby avoiding the construction of an infinite hierarchy of metamodels. It is called Meta Object Facility (MOF) 2.0 [OMG06] and represents a modeling language for the definition of modeling languages. Other approaches for defining meta-modeling languages are reviewed by Voelter [VSC06] and one specific approach (the de-facto standard approach) is Ecore [Gro09, SBPM09], which is a light-weight derivative of basic MOF concepts implemented in the Eclipse Modeling Framework (EMF).

Model Transformations

The vision of model-driven approaches is to manipulate models with *model transformation techniques* and, for example, automatically generate artifacts from models in a reliable process (depicted in Fig. 5).

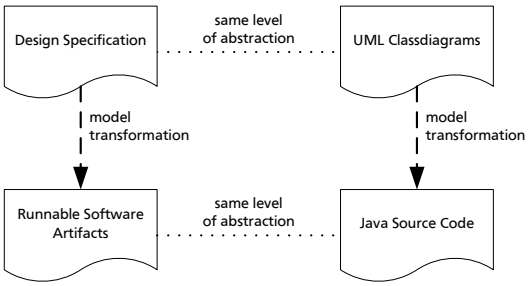


Figure 5: Condensed process of MDA aligned with the example of developing a Java program

A *model transformation* (short: transformation) is the process of taking one model as input, analyzing its content and producing an appropriate output. In this sense code artifacts are also treated as a (domain-specific) model.[2] Typically, transformations are characterized by numerous properties [CH06]. Such exemplary properties are:

- **Model manipulation**: *Inplace* transformations change the input model, which will also be returned, while *outplace* transformations read only from the input model and generate a new output model.
- **Transformation type**: Transformations can either be *unidirectional* or *bidirectional*. A unidirectional transformation is, for example, able to accept a UML class diagram and produce the corresponding Java source code artifacts. In addition, a bidirectional trans-

2 In theory, m-to-n model transformations are possible, but this thesis focuses on model transformations, which process one input and one output model only.

formation may additionally accept Java source code as input and produce the appropriate UML class diagram as output from the same specification.[3]

- **Application mode**: Model transformations can either be performed in a *batch* or *incremental* manner. Batch model transformations accept an input model and produce the appropriate output model always from scratch, while incremental model transformations accept both input and output model and a set of changes and are able to *synchronize* the models accordingly. Synchronization stands for the process of propagating changes of one model into appropriate adjustments of the opposite model.

Altogether, an actual model transformation is always a mixture of these (and other) properties. Using model transformations in the context of model-driven software development seems to be a promising means for increasing efficiency and the quality of the output. At the same time, questions concerning the relationship between input and output models, or the applicability of such instruments (e.g., which models can be efficiently processed?) have to be answered.

Range of Influences of Model Changes

A particular challenge of model synchronization (i.e., incremental model transformations) in general is to determine *precedences* between elements correctly. Therefore, a `precedence` denotes a relationship between pairs of model elements stating that one element has to be processed before another.

Consider an inheritance relationship between two classes of our example has been changed. Potentially, all attributes within the class diagram may be influenced by this model change. Actually, only those attributes of the changed class and its subclasses are affected on this model change. In practice, it is not a trivial task to differentiate between *actually affected* elements and *potentially affected* elements.

Actually affected elements are those elements in a model that have to be processed in order to propagate the applied changes appropriately to the corresponding elements in the opposite model to regain consistency. Furthermore, the set of *directly modified elements* is a subset of all *potentially affected* elements. The latter elements are also referenced as

3 Bidirectional transformations can be applied in *forward* and *backward* direction. Consider a specified bidirectional transformation between two arbitrary models A and B. A forward transformation would accept an instance of model A as input and produce an appropriate instance of B and vice versa. As we typically read from left to right, the left-hand side model of a transformation is also called the *source* model, while the right-hand side model is referred to as the *target* model. For bidirectional transformations this may cause some confusion, as in the backward transformation case the target model is the input model and the source model is the produced output model.

dependent elements. Hence, we know that directly modified elements $\subseteq$ potentially affected elements $\supseteq$ actually affected elements.

Therefore, certain heuristics have to be established to guarantee that (i) a set of all actually affected elements has been identified and (ii) this set contains as few as possible potentially affected elements that are not actually affected. Figure 6 depicts this situation graphically: The overall dependencies are implied by the triangular shape of the model. Inside the model, one change has occurred inducing a subtree of influence of all actually affected elements (shaded). A first assumption may estimate that all elements within the dotted area are affected, but this is an overestimation and, hence, the dotted area denotes the set of potentially affected elements. It is important for two reasons to to determine the set of (potentially) affected elements as close as possible compared to the actual range of influence: (i) Efficiency can be improved when less elements have to be processed and (ii) more information can be preserved that is either an additional model element or attribute value not covered by the transformation specification or reflects a rule selection during the transformation to process a node. In practice, such information is for example annotations or comments or specific (manual or heuristic) design decisions during a transformation run.

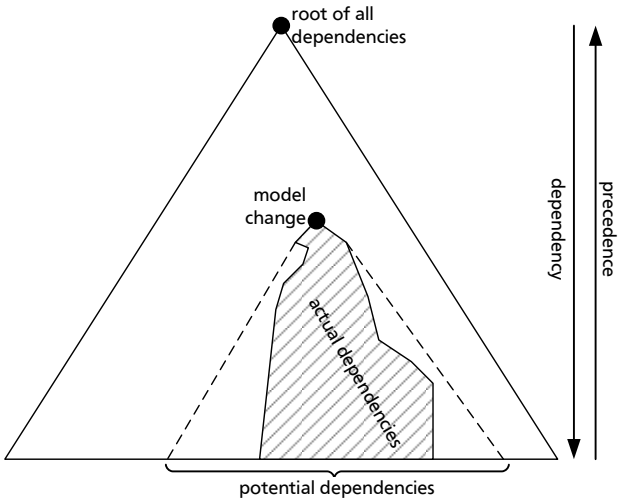


Figure 6: Model change dependencies with actually affected (shaded) and potentially affected (dotted) elements

Vertical and Horizontal Model Transformations

Model-driven software development typically utilizes *vertical* unidirectional transformations, as specifications are step-wise refined in a top-down manner until source code artifacts are produced. Unfortunately, in some scenarios this may not be sufficient. Such a scenario

is, for example, the development of a product which consists of parts that altogether form the final product. Due to this interleaving, the models used to build the parts have to be compliant to each other.

Example

Consider a Java program has to be developed, whose runtime data will be persisted in a relational database. Furthermore, consider the database development is achieved via a similar step-wise refinement starting from a database schema and producing SQL queries to access and maintain the designed database. Altogether, two processes with vertical transformations are established. New challenges arise when it becomes obvious that the models of both development processes have to be compliant to each other. The vision at this point is to introduce *horizontal* and bidirectional model transformations in order to ensure *consistency* between both domains as depicted in Fig. 7.

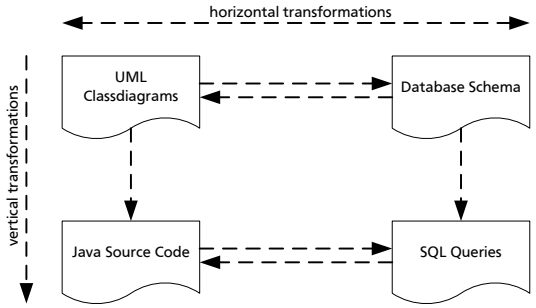


Figure 7: Parallel development of a Java program and a relational database utilizing vertical and horizontal model transformations

Consistency between models denotes a state where neither structural nor behavioral contradictions exist.[4]

In this thesis, the main foci are the application and theory of bidirectional model transformations and, more specific, bidirectional, incremental model synchronization.

CHALLENGES OF BIDIRECTIONAL MODEL SYNCHRONIZATION

As bidirectional model synchronization shall be achieved between arbitrary models, it is necessary to be aware of the challenges which are connected to this task.

- **Consistent specification**: Models have to be synchronized in forward as well as in backward direction. Hence, rules for both

4 Of course, models may contain additional (unrelated or even contradictory) data, which has been added manually besides the actual model transformation process. Nevertheless, regarding the model transformation itself, such models are still considered consistent.

forward and backward transformations have to be consistently specified.

- **Efficiency**: Model synchronization is an essential part of model-driven development. Therefore, it seems reasonable that models will be synchronized quite frequently. Efficiency is a crucial point to provide a useable approach and, hence, a synchronization should perform as fast as possible.
- **Non-determinism**: A transformation specification may allow a set of consistent results for the very same input due to specific rule selections based on **heuristic data or user interaction** during a transformation run.
- **Information preservation**: Users may introduce during the life time of a model additional information such as model elements, attribute values, or decisions regarding the application of rules during the actual transformation process that are not covered by the transformation specification. Such information shall be preserved in order to qualitatively improve the result of model synchronization and, therefore, reduce the amount of manual (unnecessary) user interaction after or during the synchronization process.
- **Expressiveness**: The favored model synchronization approach has to be applicable to practical real-world integration scenarios. Hence, the chosen transformation language must have enough expressive power in order to allow for complex rules.
- **Formal properties**: Finally, model synchronization must behave according to reliable and predictable rules. This can be achieved by complying to formal properties. Essential properties are (i) to create only consistent models and (ii) to be able to process all these models.

The vision and aim of this thesis is to improve an existing bidirectional model transformation approach in order to allow for consistent, efficient and information preserving model synchronization.

In order to achieve this, we will use a bidirectional graph transformation approach named *Triple Graph Grammar* (TGG) [Sch95]. TGG is used to declaratively specify structural consistency relationships between two models. As the term "triple" foreshadows, a third so-called *correspondence model* comes into play that represents explicit traceability links that connect elements of both models.

The TGG approach is founded on algebraic graph transformation theory [EEPT06]. Furthermore, using TGGs guarantees formal properties, such as "only consistent models are produced" or "consistent models can be processed". Besides a specification of the structure of the integration TGGs consist also of a set of *TGG rules* that declaratively express how a consistent graph triple is composed.

Such rules are used by an additional *TGG control algorithm* that operates on the provided input either in batch mode or incrementally. This control algorithm has to fulfill two tasks at runtime:

(i) Determine an appropriate processing order of the input model elements.

(ii) Select the appropriate rule for the transformation of the selected input model elements into output model elements.

In this thesis we rely on previous research of [KLKS10] that tackled task (ii) of selecting an appropriate rule sufficiently and assume that this algorithm always selects the appropriate rule (automatically or via user selection). Therefore, we put all efforts into solving task (i) sufficiently especially with having the described challenge in mind to determine the range of influence for incremental model transformations appropriately.

If we consider TGG approaches from various research groups, then more or less all challenges are already solved but mostly on their own. Regarding incremental TGG approaches, they either guarantee formal properties [HEO+11], but are inefficient or too restrictive regarding the supported expressiveness, or are efficient (and to some extent information preserving) [GH09, GPR11], but do not consider formal aspects.

Consequently, this thesis proposes a novel TGG dialect to show that both worlds can be brought together and, therefore, formal properties and efficient model synchronization can be unified in a single approach.

CONTRIBUTIONS OF THIS THESIS

Hypothesis
The aim of this thesis is to show that incremental model synchronization can be achieved with TGGs efficiently, while retaining as much information as possible. Furthermore, this solution has to be efficient and to guarantee the formal properties introduced in [Sch95, SK08].

Concrete contributions of this thesis are as follows:

- We introduce a so-called **precedence analysis** in Chap. 6 which builds the fundament of efficient and information preserving model synchronization. In addition, this analysis comes with static specification failure detection capabilities.
- Based upon the precedence analysis, we develop the core **batch transformation algorithm** in Chap. 7. This algorithm utilizes in-

formation from the precedence analysis to determine an appropriate traversal order through the input model.

- Before extending the batch algorithm with incremental capabilities, we discuss formal aspects of propagating deletions and extend the TGG theory regarding the rules with a novel concept, namely **inverse rules**, which revoke the results of earlier rule applications (cf. Chap. 9).
- The heart of this thesis is the **incremental model synchronization algorithm** proposed in Chap. 10 that uses all the previously introduced concepts.
- As no approach is perfect right from the start, we present in Chaps. 12–14 different extensions to **improve efficiency, information preserving capabilities, and expressiveness** even further.

In summary, the bidirectional model synchronization control algorithm presented in this thesis, is capable of detecting potentially affected elements due to model changes and, hence, deduce a traversal order to propagate these changes appropriately. This algorithm will work in three phases, which are (i) propagating deletions, (ii) preparing affected elements of additions, and (iii) transforming all unprocessed model elements. Finally, it will be shown that the modifications and optimizations of the TGG approach introduced in this thesis preserve all formal properties of the original TGG approach as presented in [Sch95, SK08].

Structure of this Thesis

The organization of this thesis is visualized in Fig. 8.

Part i The need for incremental bidirectional model transformation is motivated with an industrial use case scenario, TGGs are introduced from a formal point of view and it will be discussed how other approaches are or could be used to incrementally synchronize models.

Part ii Introduction of the precedence analysis and the batch control algorithm as the foundation of our incremental approach.

Part iii As incremental model synchronization requires to revoke the effects of former rule applications, TGG theory is extended with an appropriate concept and, finally, the incremental control algorithm is presented.

Part iv Various extension points are discussed to further improve the precedence-driven approach.

Part v This part provides a discussion of implementation details of our TGG approach together with an evaluation.

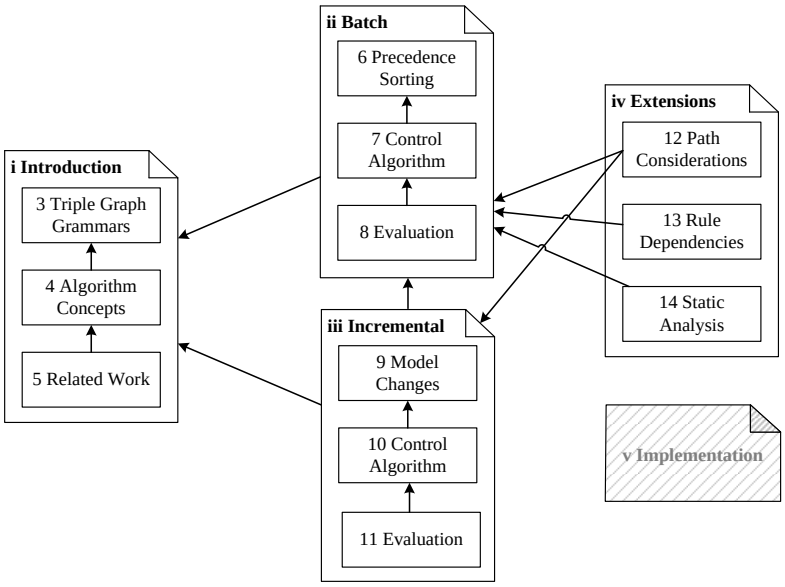


Figure 8: Dependencies between the contents of the different parts

Part vi The thesis is concluded with a summary and outlook of how to extend and improve the proposed approach even further.

Parts ii–iv are completed with an evaluation to highlight the benefits and drawbacks of the details presented in each part.

For all parts, the same conventions are used to support readers in understanding and extracting the core concepts, important aspects and ideas:

- *Italics* are used to highlight the first occurrence and the definition of an important term.
- In addition, an index is provided in the end of this document that helps readers to easily find explanations and definitions of referenced terms.
- Every term written as `term` refers to an element of a diagram, figure, or source code fragment.
- The thesis is written in "mixed-mode", which means definitions and running examples are intertwined. This style of presentation is considered to ease access to complicated definitions and, hence, readers may understand definitions and underlying concepts more easily.

2 MOTIVATION

This chapter illustrates how real-world application scenarios may benefit from bidirectional and incremental model transformation approaches. Although this example will not accompany us throughout the thesis, it is able to highlight numerous requirements for model transformations and provides us with an additional feeling regarding challenges in this field.

We start with a short overview on systems engineering, before discussing a complex case study from an industrial research cooperation which motivates the demand for model synchronization in an industrial context.

2.1 SYSTEMS ENGINEERING

According to [Wei06, SR09], a system is a collection of components which are designed to achieve a common goal, which could not be accomplished from individual components on their own.

Typical examples for systems are air planes, plants, space crafts, or robots. Obviously, each system is built of numerous components, which offer their functionality to the overall system. Consider a space craft for example, where complex maneuvering engines have to be developed. Such engines have to be powerful and reliable enough to navigate the space craft along its designated track. But without sophisticated and well-tested software, the sensor input cannot reliably be interpreted as an input for computing necessary maneuvering parameters.

Both components, the engines and the control software, have been developed according to appropriate process models such as the previously described `V-Modell XT` for software engineering. In addition, systems engineering has to govern the overall process of how to integrate the distinct components to the final product. Therefore, systems engineering describes the process of building products, which consist of various hardware and/or software components. Systems engineering is a complex task that involves a number of different engineering disciplines to be coordinated while creating collaboratively a product. By definition, systems engineering integrates all involved disciplines [Wei06] into building the final product.

2.2 model-driven automation engineering

In a research cooperation with Siemens Industry Automation we investigated between November 2008 and October 2010 model-driven techniques to support engineering of complex plants and machinery (i.e., automation engineering). As described previously, engineering plants and machinery belongs to the field of systems engineering. Beyond prototypical implementations [LSRS10], we elaborated different requirements such as relying on formally founded approaches or necessary language features for model transformation definitions.

This section subsumes the elaborated results and describes which requirements are to be met by model-driven technologies, and especially by incremental model synchronization, to be applicable in real-world application scenarios such as automation engineering. Therefore, this section originates from the ideas and concepts developed for publications [SRS09, LSRS10, SLRS10, RLSS11] in this context.

2.2.1 *Automation Engineering Scenario*

The development of automation systems for machines and plants depends on information from an increasing number of engineering tools like mechanical CAD, automation device configuration or control logic engineering. Automation engineering of programmable logic controllers (PLCs) requires for example information about the devices used for machine automation, their characteristics, and their interaction with other machine modules from other engineering tools like electrical engineering or mechanical engineering software.

Since information exchange between these tools and design models is mostly based on design documents and meetings, there is a strong requirement for a tighter integration of PLC engineering models to increase design efficiency. Automation system providers like machine builders drive the integration of PLC engineering by the establishment of mechatronic development processes which shall integrate mechanical engineering, electrical engineering and automation engineering [VDI04a]. Automation system users like automotive companies investigate how to realize the digital factory [VDI04b] with engineering models available for designing, commissioning and operating production sites.

Model-Driven Automation Engineering (MDAE) [RLSS11] addresses the requirement for the integration of PLC engineering with other disciplines and establishes a bidirectional synchronization between the design models used for the development of automation systems. In this context, the term PLC refers to both hardware and software technologies and not only to a special hardware architecture (e.g., in

contrast to PC-based soft logic controllers). The software technology of PLC controllers is defined by the standard IEC 61131 [IEC03].

Machine development is based on an established development process of machine builders in an existing tool environment. Therefore, the introduction of an environment with model exchange between these engineering tools is usually an a-posteriori integration of existing tools and tool interfaces. As an automation application example, we look at the automation of a storage and retrieval machine of a high-bay warehouse system based on the tools Comos ET [Sie12] and Simatic Step7 [Sie10]. The storage and retrieval machine runs within a warehouse aisle (see Fig. 9 right-hand side) and picks or places goods from the storage shelf.

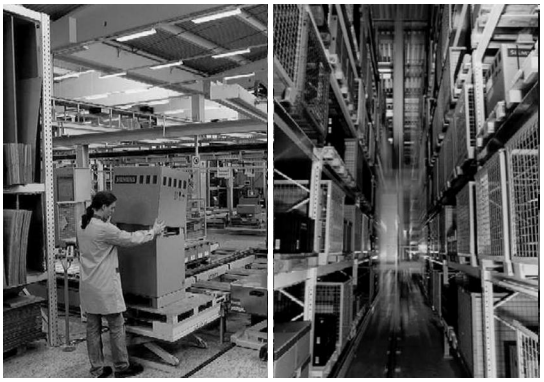


Figure 9: High-Bay Warehouse [LSRS10]

The overview in Fig. 10 shows the automation devices used in such a high-bay warehouse system example: a Siemens Simatic `CPU 317T-2 DP` controller with integrated motion control functions, distributed I/O (input/output) modules with Siemens Simatic `ET 200S` and motor control by a Siemens Sinamics drive system. Complex components like data matrix systems (e.g., for identification of goods at the commissioning, left-hand side of Fig. 10) or handling robots are connected by fieldbus communication.

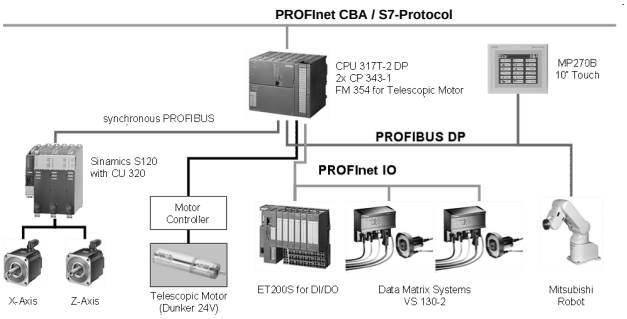


Figure 10: Exemplary hardware of a storage and retrieval machine [LSRS10]

2.2.2 *Change Propagation Workflow Example*

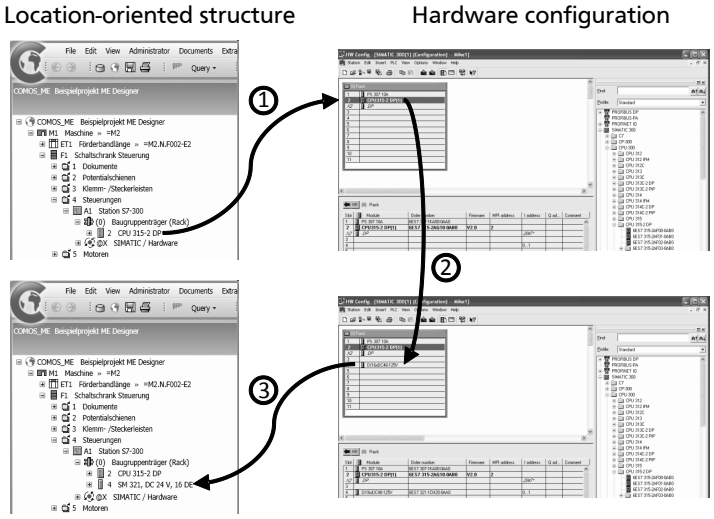
The detailed integration workflow used in this scenario is shown in Fig. 11. The previously described information is stored in our scenario in two different models: (i) the location-oriented structure represents the physical composition of a specific plant and (ii) hardware configurations describe the logical interconnections between these components.

The location-oriented structure has been specified in the IEC standard 61346 [IEC96] and its successor IEC 81346 [IEC09]. Hence, the plant is decomposed into different components which may contain each other or may be physically wired via ports. This information is modeled in the tool Comos ET (left-hand sides of Figs. 11(a) and (b)). The right-hand side of Fig. 11(a) and (b) depict screenshots of the Simatic Step7 tool. Within this tool, hardware centric information (i.e., a hardware configuration) is modeled which is further used to program a PLC. Such a model contains abstract information e.g., regarding the type of a processor or communication model while neglecting actual physical details such as the actual size or weight. Furthermore, these modules may communicate via logical connections which do not necessarily reflect the actual physical wirings as described in the location-oriented model.

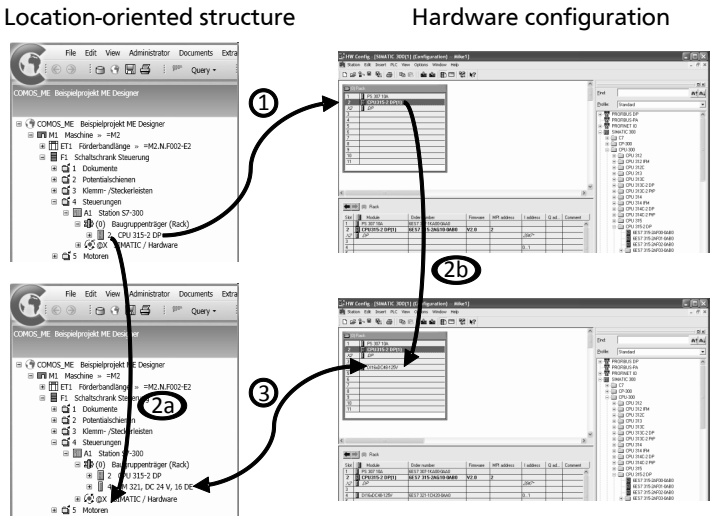
The purpose is to synchronize location-oriented structures with hardware configurations. Hence, the initial location-oriented structure of a new machine is created in the tool Comos ET. This data shall be propagated to the tool Simatic Step7 (Step 1 in Fig. 11(a)). For this propagation step the model of the location-oriented structure is the source model (input model) of the data propagation.

The data propagation can be implemented with a model transformation because information in the source model (i.e., the location-oriented structure) has to be transformed into adequate information of the target model (i.e., the Step7 hardware configuration). Since no Step7 project exists yet, an initial Step7 project is created with the single PLC visible in the hardware configuration. This batch transformation decreases the effort of initiating a project.

With that project at hand, the automation engineer starts developing the hardware configuration. In our application scenario, the automation engineer adds for example an additional I/O module. Therefore, this additional I/O module is added to the hardware configuration of Step7 (Step 2 in Fig. 11(a)). Next, as the changes induce an inconsistent state between the two integrated models, these changes have to be propagated into the appropriate location-oriented model in Comos ET. Hence, an additional I/O module is also created in the Comos ET project (Step 3 in Fig. 11(a)). In this synchronization step, the roles of source and target models are swapped: the



(a) Sequential Integration Workflow [LSRS10]



(b) Parallel Integration Workflow

Figure 11: Integration workflows between location-oriented structures in Comos ET (left) and hardware configurations in Step7 (right)

automation engineering model is the source model of the data propagation and the model of the location-oriented structure is the target model. This will be achieved with an incremental model synchronization technique.

Another typical workflow is depicted in Fig. 11(b). Again, the location-oriented structure is created in the Comos ET tool and afterwards all necessary information is propagated to the Step7 tool. In contrast to the previous use case, now both models evolve in parallel (Steps 2a and 2b in Fig. 11(b)). Finally, both models have to be updated to a consistent state. This has to be achieved by an incremental change-aware model synchronization technique, that allows for concurrent modifications (Step 3 in Fig. 11(b)).

In both scenarios, the transformation process has to regard the specific dependencies in these models to determine an appropriate propagation sequence in order to ensure that all changes are propagated efficiently and, secondly, that information is preserved as much as possible.

2.3 REQUIREMENTS OF A MDAE MODEL SYNCHRONIZATION

Using model transformation techniques in order to cope with the synchronization task in the field of MDAE induces various requirements which have to be fulfilled by an appropriate transformation approach. The described requirements are partially adapted and extended from [LSRS10].

In the development process of such a high-bay warehouse system, the configuration of the Simatic I/O modules (shown in the automation structure in Fig. 10) changes if an engineer adjusts the wiring of the devices built in electrical cabinets. In the opposite direction, changes of I/O modules in Step7 due to programming requirements must be reflected within the location-oriented structure as well. This engineering workflow is a basic use case of MDAE. Both incremental change propagation and batch transformation are useful, as not only existing models must be updated consistently in an incremental manner, but also newly created projects could be used to create models with a batch transformation to decrease the initial efforts in setting up a model (cf. Sect. 2.2.2).

Depending on the responsibilities in an engineering organization, automatic change propagation might not be allowed due to legal or organizational restrictions. Instead notifications about inconsistencies between the engineering models should be generated. Resolving these inconsistencies remains within the responsibilities of the engineers of each discipline. The implementation of consistency checks is usually easier than change propagation, since the actions required for reconciliation need no implementation.

Both change propagation and consistency checks may use *traceability* links between related elements of different engineering models. *Traceability* in this regard stands for specific information that encodes which elements in the two models are related. Thus, traceability links constitute a specific model (i.e., trace or correspondence model) that explicitly relates elements of integrated models. With such an additional trace model at hand it is possible to identify related elements, e.g., the hardware component for which a specific I/O module is used for and, hence, identify the source for (potential) model changes.

As a non-functional requirement, the model synchronization approaches, should be usable from within an existing engineering tool environment and should not presume the introduction of new engineering tools. Usually machine builders have an existing tool environment as described in Sect. 2.2.2 but with manual data exchange between these tools or hand-crafted unidirectional batch data exchange solutions [Hof11].

List of Requirements

As we have seen, real-world transformation scenarios such as in the domain of MDAE have complex preconditions and expectations. The following list summarizes the requirements in alphabetical order with a (short) description for each requirement.

- *Bidirectionality*: A model transformation specification can be unidirectional or bidirectional. While unidirectional transformation specifications always own a designated direction (e.g., transformations are only possible from a location-oriented structure to a hardware configuration but not in the other direction), bidirectional transformation specifications define implicitly both directions. Regarding our usage scenario, transformations in both directions are needed. Bidirectionality is important due to the fact that engineering processes are typically not sequential. Therefore, it is of vital interest that decisions and changes can be propagated in both directions.
- *Concurrency*: Integrated models may evolve in parallel to additionally speed-up the production phase. Typically, this may induce contradictory model changes that have be consistently synchronized either automatically or with user guidance. Nevertheless, a fully-fledged transformation framework should support concurrent model changes.
- *Declarative specifications*: A model-to-model transformation specification can be expressed in an imperative or a declarative manner. An imperative specification describes the complete model manipulation process of the actual transformation. In contrast, by using declarative means, the result of a transformation is speci-

fied and not the way how to get there. The engineers specifying such transformations describe how a fully integrated model looks like and can rely on underlying formal techniques that this specification can cope with any kind of valid input. Declarative specification saves time and space because most imperative information can be directly derived. Instead of describing the evolution of a model in all facets (e.g., creating new elements or propagate attribute value changes), only consistent models have to be described.

- *Efficiency*: Any applied model transformation technique should be able to fulfill its task within acceptable ranges of time and resource consumption. Regarding systems with restricted memory resources, the transformation technique must be able to stick to these limits. Regarding the interaction with users, the transformation technique should deliver a result in an appropriate amount of time as this is a basic feature of usability [Nie94].
- *Incrementality*: Changes in models have to be recognized and appropriate actions must be triggered. Incremental model synchronization is one of the major demands in real-world applications: It is widely known that models do not live without any changes; in contrast, models develop in complex scenarios. Especially when large model have to be integrated and/or models with user-specific changes that cannot be re-created, it is of major importance that the corresponding model can be updated with the appropriate set of changes instead of being re-computed from scratch.
- *Information preservation*: Whenever a model change has to be synchronized with the opposite model, only actually affected elements should be considered. This requirement coincides with the requirement of efficiency but adds additional properties: Considering a model synchronization, a naïve control algorithm could remove the whole target model and create a new model from scratch. The result would be consistently integrated models. Typically, users introduce additional information to their models which would have been dropped in this case. Such information is for example some parts of a model that are not covered by the specified transformation. Another type of such information is explicit decisions during a transformation process regarding which rule should be applied for processing a certain element (see requirement "non-determinism" below). In both cases such information shall be retained to reduce the amount of information that has to be manually re-introduced during or after the synchronization and, therefore, directly decrease the workload of the synchronization user.

- *Non-determinism*: In practical scenarios it is often necessary to introduce a certain degree of freedom in the model transformation specification to reflect specific design decisions. Consider for example a `CPU` in the location-oriented structure which may be used in the hardware configuration to operate as a single processor or as a backup processor. At runtime, a control algorithm could retrieve additional *input from the user* or use a predefined heuristic to decide how to process an actual `CPU`. Such a transformation specification is called *non-deterministic* as a variety of consistent pairs of models are possible. In combination with information preservation (see above), such decisions should be preserved in order to avoid unnecessary re-calculation or user queries.
- *Traceability*: Reliability and responsibility are major aspects of any processes building a deliverable system. Since many different models, modeling activities and engineering domains are involved in building a system it is of major importance that at any point in time specific properties of a model can be traced to their origins. In order to produce reliable and trustworthy systems it is mandatory that any decision in the process is traceable. Regarding the application example with traceability links between corresponding physical wirings and logical connections, engineers are enabled to understand why certain logical or physical connections have been established. If such connections have to be adjusted due to any reasons, it is easily possible to determine all affected wirings or connections in the other model and update them appropriately.
- *Validation*: We demand that integration approaches do not only "work on a paper base" but in real systems. The model synchronization technique should be used successfully in different domains and, therefore, empirically show its capabilities for real-world application scenarios.
- *Verification*: The purpose of model-to-model transformation approaches is to create and modify model data. In order to ensure that such modifications are always correctly applied and never in a harmful manner (e.g., turning a correct model into an incorrect model), the application logic and semantics of transformation rules must be verified. Formal verification proves certain concepts and aspects of the integration technique.

3 TRIPLE GRAPH GRAMMARS

The previous chapters revealed basic requirements, ideas and concepts to tackle the task of bidirectionally synchronizing models. We investigated these demands and prerequisites using a real-world industrial application example from an industrial cooperation [SRS09, LSRS10, SLRS10, RLSS11]. Next to this, other applications in this area seem to profit from bidirectional model synchronization [KLS+12, SK12] as well.

Nevertheless, without having discussed the actual implementations of these examples, it became obvious that such scenarios are not usable as an easy-to-handle running example for two reasons:

- The size of the actual transformation is too large. This may cause unreadable specifications which are in addition hard to remember throughout the rest of this thesis.
- The concepts used in the actual transformation are quite homogeneous, which means that typically one element is extended with a new subtree of new elements.

In order to describe the introduced concepts, benefits and drawbacks, an example is required that is (1) small regarding the size of the specification and (2) heterogenous regarding the structure of the specification. Finally, the application scenarios of Chap. 2 massively rely on attribute comparisons. As attribute comparisons were out-of-scope for this thesis,[1] we decided to restrict our running example to structural aspects only.

3.1 GRAPHS AND TYPE GRAPHS

The formal concepts described in this chapter are adapted from Ehrig et al. [EEPT06], in which fundamental approaches of graph transformation theory are formalized, and from Klar et al. [KLKS10], in which basic concepts for TGGs have been revisited, refined and extended (compared to the original publication of Schürr [Sch95]).

Example

The running example used throughout the rest of this thesis is an imaginary integration between a `Company (structure)` and an `IT`

1 Note that this research topic was successfully tackled in [AVS12] and, thus, is part of our recent TGG implementation.

(infrastructure) (short: IT). The idea is to assign each Employee of a Company either a desktop computer (PC) or a mobile computer (Laptop). Furthermore, Admins take care for Routers and Networks. A concrete example is depicted in Fig. 12, where a small Company is controlled by a CEO who hired one Admin and two Employees.

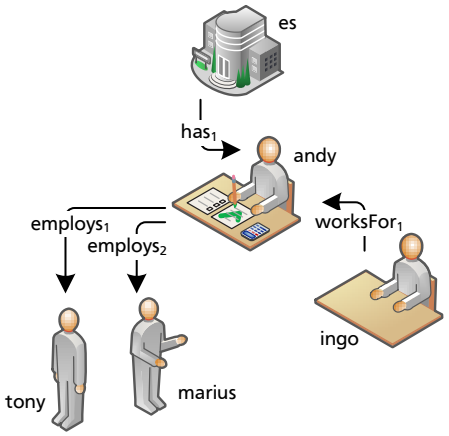


Figure 12: Concrete instance of a Company named es

Each Employee receives either a PC or a Laptop to fulfill his designated tasks. In addition, these computers must be connected via an internal Network controlled by a Router. The appropriate infrastructure is depicted in Fig. 13.

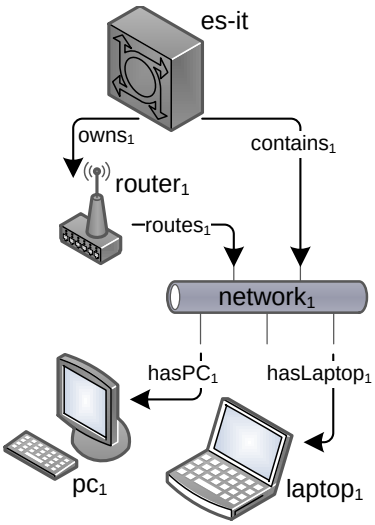


Figure 13: Concrete instance of an IT structure named es-it

Obviously, these two concrete instances of a Company and an IT consist of elements that belong together: it takes an Admin to maintain the Router and the Network, while the Laptop and PC are likely to be used by an Employee. These relationships are depicted in Fig. 14.

Figure 14 gives us an impression, which elements should be consistently integrated. As, for example, an Employee needs a PC or Laptop

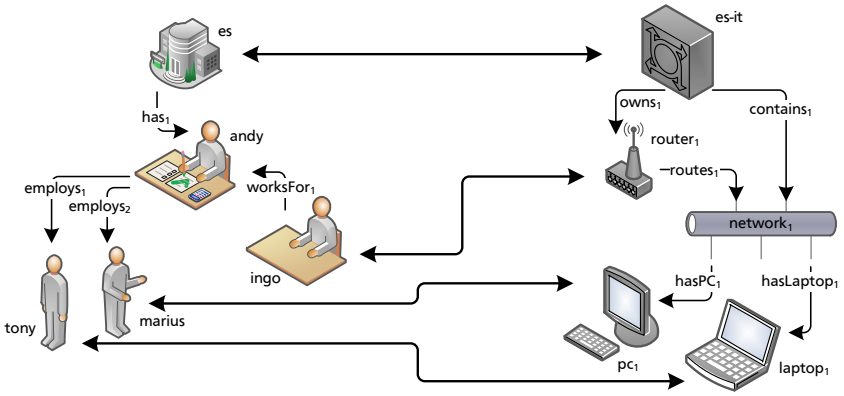


Figure 14: Concrete instance of an integrated `Company` and `IT`

to start his or her tasks, the corresponding transformation specification should reflect this appropriately. The actual application of such a specification will be referred to as *forward transformation* because it goes from the source model (left-hand side) to the target model (right-hand side). Another possible use case is that it has been decided to introduce an additional `Router` and, therefore, equip the `IT` with an extra `Network`. This directly leads to the requirement that the `Company` needs to find a suitable administrator to maintain the network. Such a process denotes a *backward transformation*.

At first, we define *graphs* and *type graphs*. The concept of graphs will be used to describe actual models while the concept type graphs denotes the type system of models and, therefore, is the metamodel of a model (cf. Chap. 1).

Definition 1 (Graphs and Graph Morphisms)
A graph $G = (V, E, s, t)$ *consists of finite sets* V *of nodes, and* E *of edges, and two functions* $s, t : E \to V$ *that map each edge to its source and target node. A* graph morphism $h : G \to G'$, *with* $G' = (V', E', s', t')$, *is a pair of functions* $h := (h_V, h_E)$ *where* $h_V : V \to V'$, $h_E : E \to E'$ *and* $\forall e \in E : h_V(s(e)) = s'(h_E(e)) \wedge h_V(t(e)) = t'(h_E(e))$.

Example

Considering our running example from Fig. 12, we see a graph with the following sets:

- $V = \{\texttt{es}, \texttt{andy}, \texttt{ingo}, \texttt{tony}, \texttt{marius}\}$
- $E = \{\texttt{has}_1, \texttt{employs}_1, \texttt{employs}_2, \texttt{worksFor}_1\}$

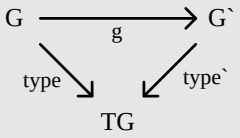
Thus, the functions s, t are defined as follows:

- $s(\texttt{has}_1) = es,\ t(\texttt{has}_1) = \texttt{andy}$

- $s(\texttt{employs}_1) = \texttt{andy}$, $t(\texttt{employs}_1) = \texttt{tony}$
- $s(\texttt{employs}_2) = \texttt{andy}$, $t(\texttt{employs}_2) = \texttt{marius}$
- $s(\texttt{worksFor}_1) = \texttt{ingo}$, $t(\texttt{worksFor}_1) = \texttt{andy}$

Next, metamodels are formalized as type graphs and models as typed graphs:

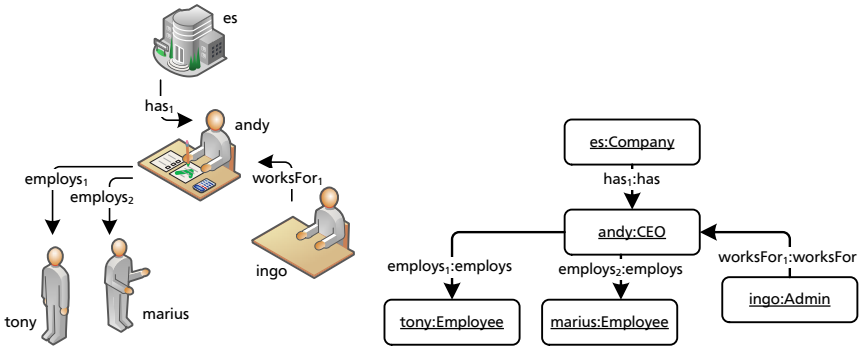
Definition 2 (Typed Graphs and Typed Graph Morphisms)
A type graph *is a graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$.
A typed graph $(G, type)$ *consists of a graph* G *together with a graph morphism* type*:* $G \to TG$.
Given typed graphs $(G, type)$ *and* $(G', type')$, $g : G \to G'$ *is a* typed graph morphism *iff the following diagram commutes:*



Thus, the set $\mathcal{L}(TG)$ *denotes all correctly typed graphs* G *over* TG.

Example

Figure 15 depicts the company structure G of our running example together with its type information in concrete and abstract syntax. Note that from now on we are switching from so-called *concrete syntax* to *abstract syntax*. This syntax normalizes all future figures and presents in a compact manner instance and type information. Names of model elements are written in lower case separated by a colon from their type in upper case. Additionally, concrete model elements are underlined. Note that edges in typed graphs are not underlined, and edge types are also written in lower case to improve readability.



(a) Concrete syntax (b) Abstract syntax

Figure 15: Exemplary `Company` structure in concrete and abstract syntax

Formally, the type graph (i.e., metamodel) TG_{Comp} for a `Company` consists of the following sets:

- $V_{TG_{Comp}} = \{\texttt{Company}, \texttt{CEO}, \texttt{Employee}, \texttt{Admin}\}$
- $E_{TG_{Comp}} = \{\texttt{has}, \texttt{employs}, \texttt{worksFor}\}$

Hence, the functions $s_{TG_{Comp}}, t_{TG_{Comp}}$ are defined as follows:

- $s_{TG_{Comp}}(\texttt{has}) = \texttt{Company}$, $t_{TG_{Comp}}(\texttt{has}) = \texttt{CEO}$
- $s_{TG_{Comp}}(\texttt{employs}) = \texttt{CEO}$, $t_{TG_{Comp}}(\texttt{employs}) = \texttt{Employee}$
- $s_{TG_{Comp}}(\texttt{worksFor}) = \texttt{Admin}$, $t_{TG_{Comp}}(\texttt{worksFor}) = \texttt{CEO}$

Finally, the morphism $type_G : G \to TG_{Comp}$ consists of two functions $type_G := (type_{G_V}, type_{G_E})$, where $type_{G_V}$ encodes the typing for nodes (i.e., vertices) and $type_{G_E}$ encodes the typing for edges as follows:

- $type_{G_V}(\texttt{es}) = \texttt{Company}$
- $type_{G_V}(\texttt{andy}) = \texttt{CEO}$
- $type_{G_V}(\texttt{ingo}) = \texttt{Admin}$
- $type_{G_V}(\texttt{tony}) = \texttt{Employee}$
- $type_{G_V}(\texttt{marius}) = \texttt{Employee}$
- $type_{G_E}(\texttt{has}_1) = \texttt{has}$
- $type_{G_E}(\texttt{employs}_1) = \texttt{employs}$
- $type_{G_E}(\texttt{employs}_2) = \texttt{employs}$
- $type_{G_E}(\texttt{worksFor}_1) = \texttt{worksFor}$

3.2 GRAPH GRAMMARS

Model transformation can be realized with graph grammars because models can be treated as graphs. Graph grammar approaches use declarative rule specifications to express the evolution of graphs. This concept seems to be quite natural and has been already widely used in string grammar definitions (e.g., EBNF). A rule specification r consists of two graphs named *left-hand side (*L*)* and *right-hand side (*R*)* and is denoted as $r := (L, R)$.

Example

Considering the running example, rule `addNewLaptop` expresses that a concrete `Network` must exist in order to add a new `Laptop` instance (depicted in Fig. 16).

Applying such a rule to a concrete model has to regard the actual model appropriately. Consider two or more `Networks` exist already: which network should be extended? The rule as depicted above only states that a `Network` must exist in order to append a `Laptop`.

A category theoretic approach named *double pushout* (DPO) has been established to formalize model transformation appropriately. A

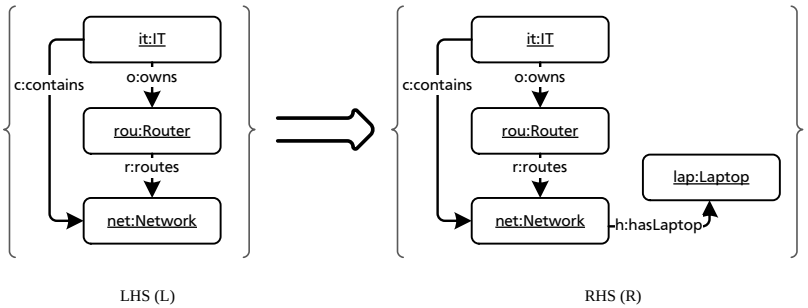


Figure 16: Informal rule to extend a `Network` with a `Laptop`

pushout describes which elements of a graph shall be matched and extended. Furthermore, the double pushout formally introduces the concept of a *gluing graph* K which is identified inside an actual model and relatively from this gluing graph elements are deleted or added (cf. Def. 4). This is achieved via morphisms (i.e., structure-preserving mappings) that map each element of a rule to a suitable element of an actual graph. When a match (i.e., a concrete subgraph in a model that complies with the pattern specified by L) is found, the DPO approach ensures that elements are added or removed to/from suitable remaining elements only. Formally, *rules* are defined as follows:

Definition 3 (Production Rules)
A typed graph production rule $:= (L \xleftarrow{l} K \xrightarrow{r} R)$ *consists of typed graphs* L, K, *and* R, *called the left-hand side, gluing graph, and the right-hand side respectively, and two injective typed graph morphisms* l *and* r.

Example

Considering our running example from above, our rule `addNewLaptop` is defined as depicted in Fig. 17.

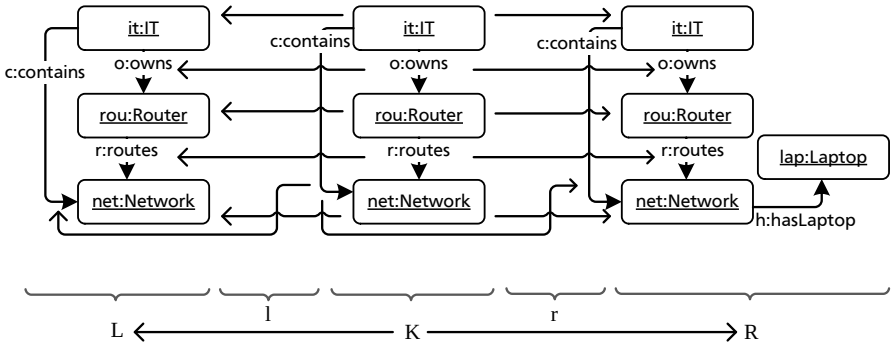
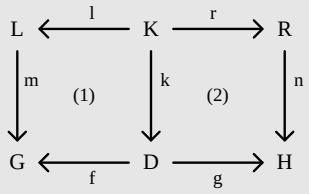


Figure 17: Formal definition L, K, R of rule `addNewLaptop`

Furthermore, the *application* of a rule is defined via pushouts as follows:

Definition 4 (Rule Applications)
Given a (typed) graph production rule $= (L \xleftarrow{l} K \xrightarrow{r} R)$ *and a (typed) graph* G *with a (typed) graph morphism* $m : L \to G$*, called the match. A direct rule application* $G \overset{r@m}{\rightsquigarrow} H$ *from* G *to a (typed) graph* H *is given by the double-pushout (DPO) diagram depicted below, where (1) and (2) are pushouts.*



Note that this rule *is only applicable to* G *iff the gluing condition [EEPT06] is satisfied. This condition demands that after deleting elements from* G *no* dangling edges *may exist. A dangling edge denotes an edge* e *in* G *without a source and target node, or with a source or target node only.*

Example

Considering our running example, a graph transformation, i.e., applying rule `addNewLaptop` on a graph G, works as follows:

1. Find a match m that represents a subgraph in G which complies with L.
2. Find the appropriate match k in G.
3. Remove all elements in G that are not in the intersection of both matches to retrieve D.
4. Extend D with additional elements such that afterwards a match n can be found.

Figure 18 depicts the whole process graphically.

Finally, we define the concept of a *graph grammar* which subsumes the concepts of typed graphs and rules.

Definition 5 (Graph Grammars and Their Languages)
A graph grammar $GG := (TG, \mathcal{R})$ *is defined by a type graph and a set of typed rules* $\mathcal{R}$*. The graph grammar language* $\mathcal{L}(GG)$ *is the set of all graphs* G_i *typed over* TG *that can be derived by applying rules* $r_j \in \mathcal{R}$*.*
Formally, $\mathcal{L}(GG) := \{G \in \mathcal{L}(TG) | G_0 \overset{r_1@m_1}{\rightsquigarrow} G_1 \overset{r_2@m_2}{\rightsquigarrow} \dots \overset{r_n@m_n}{\rightsquigarrow} G_n\}$*, where* $G_0 = \emptyset$ *denotes the empty starting graph.*

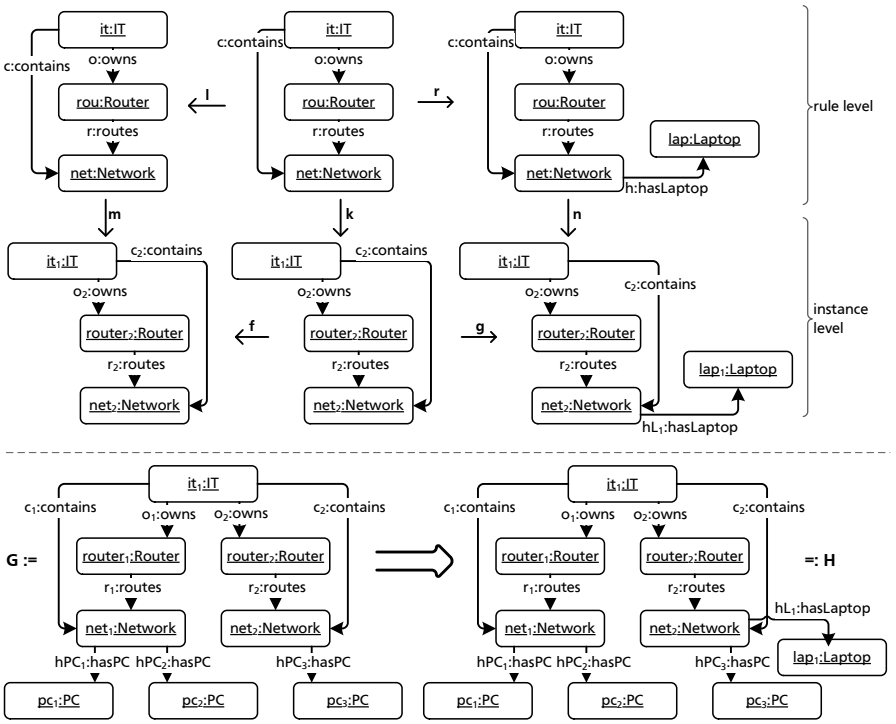


Figure 18: Application of rule `addNewLaptop` to a concrete graph G

3.3 TRIPLE GRAPH GRAMMARS

As an extension of Pratt's pair grammars [Pra71], TGGs have been introduced in 1994 by Schürr [Sch95]. The aim was to connect two source and target graphs via a third so-called *correspondence graph* in between. Of course this could also be achieved by introducing additional edges between source and target graphs, but would lead to the requirement to adjust the corresponding type graphs of them. As this is not always suitable (due to legal, organizational, or technical restrictions), the correspondence graph allows for a light-weight means to achieve this goal. This also meets the requirement of having explicit traceability links between models (cf. requirements of real-world applications in Chap. 2).

A *graph triple* consists of three graphs. Each graph is in the set of graphs of a particular language, i.e., conforms to a *graph schema* denoted by a certain type graph. In addition, two morphisms h_S and h_T relate elements of the correspondence graph with elements of the source and target graph.

Definition 6 (Typed Graph Triples)
Given the languages $\mathcal{L}(\mathsf{TG}_S)$, $\mathcal{L}(\mathsf{TG}_T)$, *and* $\mathcal{L}(\mathsf{TG}_C)$ *for the source, target and correspondence domain respectively.*
A graph triple $\mathsf{GT} := G_S \overset{h_S}{\leftarrow} G_C \overset{h_T}{\rightarrow} G_T$ *is properly typed iff (1)* $G_S \in \mathcal{L}(\mathsf{TG}_S)$, *(2)* $G_T \in \mathcal{L}(\mathsf{TG}_T)$, *(3)* $G_C \in \mathcal{L}(\mathsf{TG}_C)$, *(4)* $h_S : G_C \to G_S$, *and (5)* $h_T : G_C \to G_T$. *Note that* h_S *and* h_T *are morphisms between typed graphs.*

Example

Considering our running example, Fig. 19 depicts the typed graph triple with source, correspondence and target elements expressing the relationships as described in the beginning of this chapter. Note that previously omitted correspondence elements are now depicted explicitly.

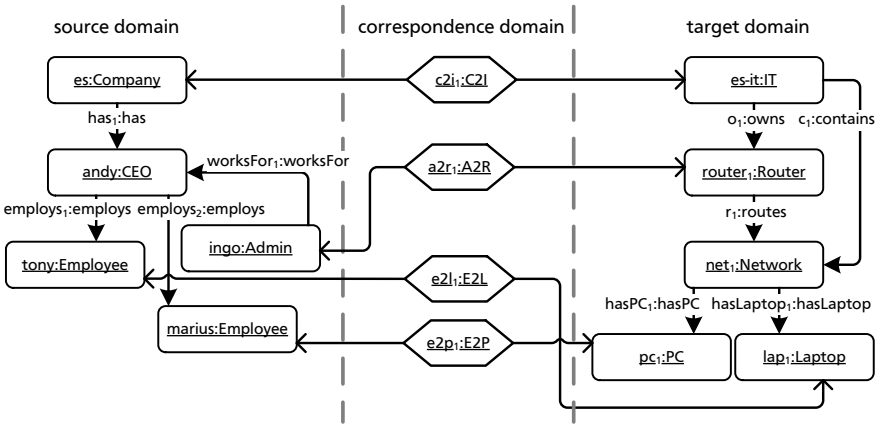


Figure 19: Typed graph triple of our running example

Furthermore, the type graph triple is depicted in Fig. 20. Our running example specifies the integration of company structures and corresponding IT structures. The *TGG schema* is the type graph triple for our running example. The *source domain* is described by a type graph for company structures: A `Company` consists of a `CEO`, `Employees` and `Admins`. In the *target domain*, an `IT` provides `PCs` and `Laptops` in `Networks` controlled by a `Router`. The *correspondence domain* (center) specifies links between elements in the different domains. Note that the edges from the correspondence to the source and target domains denote morphisms and not instances of specified edge types. Hence, these edges are displayed without names and types.

According to Def. 2, we have to define what a type preserving graph triple morphism is:

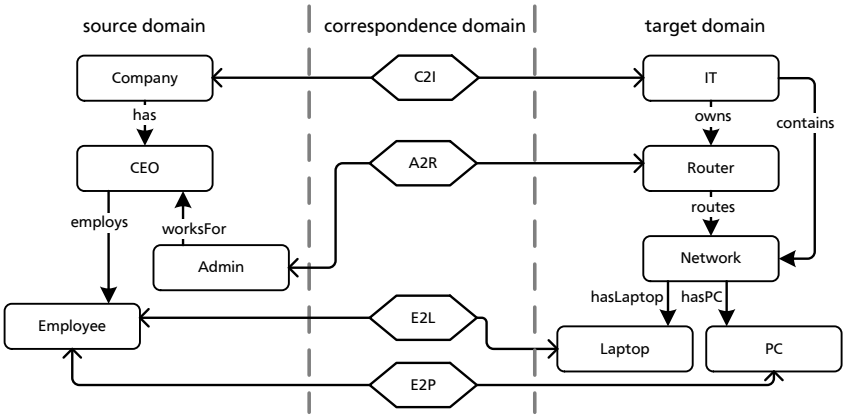
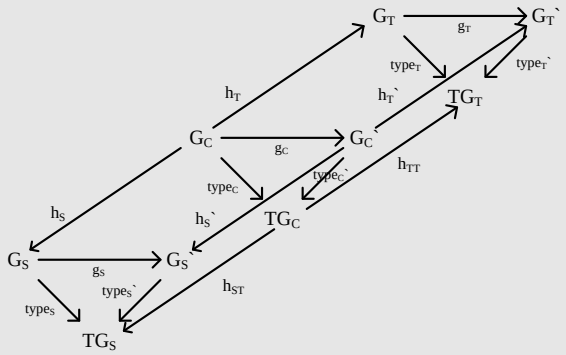


Figure 20: Type graph triple, also know as TGG Schema

Definition 7 (Type Preserving Graph Triple Morphism)
A typed graph triple $(\mathsf{GT}, \mathsf{type})$ *consists of a graph triple* GT *together with three morphisms* $\mathsf{type} := (\mathsf{type}_S, \mathsf{type}_C, \mathsf{type}_T)$*, where* $\mathsf{type}_S : G_S \to TG_S$*,* $\mathsf{type}_C : G_C \to TG_C$*, and* $\mathsf{type}_T : G_T \to TG_T$ *such that* $\mathsf{GT} \overset{\mathsf{type}}{\to} \mathsf{TGT}$ *and* $\mathsf{TGT} := TG_S \overset{h_{ST}}{\leftarrow} TG_C \overset{h_{TT}}{\to} TG_T$*. A graph triple morphism* (g_S, g_C, g_T) *with* $g_S : G_S \to G'_S$*,* $g_C : G_C \to G'_C$*, and* $g_T : G_T \to G'_T$ *is type preserving iff the following diagram commutes:*

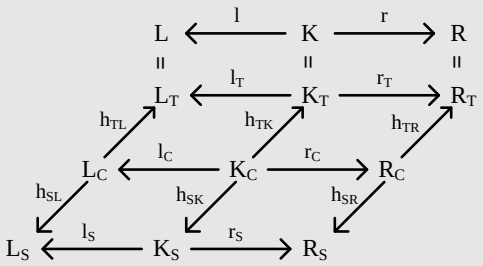


Hence, $\mathcal{L}(\mathsf{TGT})$ *denotes the set of all correctly typed graph triples* GT *over* TGT*.*

We have now defined triple graph morphisms. The next step is to specify how a *triple rule* is applied.

Definition 8 (Triple Graph Production Rules)

A typed triple graph production rule rule $:= (L \xleftarrow{l} K \xrightarrow{r} R)$ *consists of typed graph* triples[a] L, K, *and* R, *and two injective type preserving graph triple morphisms such that the following diagram holds:*



a Note that L, K, R are from now on representing triples of graph.

As TGGs are monotonous regarding the specification of rules, it is prohibited to delete elements in a declarative rule. Monotonicity in this sense means that rules may only add new elements.[2] Therefore, L and K are identical for any TGG rule r, which means that no elements are deleted when applying this rule. Hence, the definition of rules can be simplified which is achieved in Def. 10.

Elements in L denote the precondition of a rule and are referred to as *context elements*, while elements in $R \setminus L$ are referred to as *created elements*.

Example

Together, with our TGG schema (cf., Fig. 20) we can now specify the actual triple graph rules for our running example. The rules depicted in Fig. 21 build up an integrated company and IT structure simultaneously. Rule (a) creates the root elements of the models (a `Company` with a `CEO` and a corresponding `IT`), while Rule (b) appends additional elements (an `Admin` and a corresponding `Router` with the controlled `Network`). Rules (c) and (d) extend the models with an `Employee`, who can choose a `PC` or a `Laptop`. We use a concise notation by merging L and R of a rule, depicting context elements in black without any markup, and created elements in green with a "++" markup.

2 At a first glance it may seem inappropriate that TGG rules are only allowed to add elements as a synchronization algorithm also has to cope with deletions. In practice, this is no problem as these rules are not actually applied but used for deriving so-called *operational rules*, which also cope with the deletion of elements. This will be explained later in this chapter.

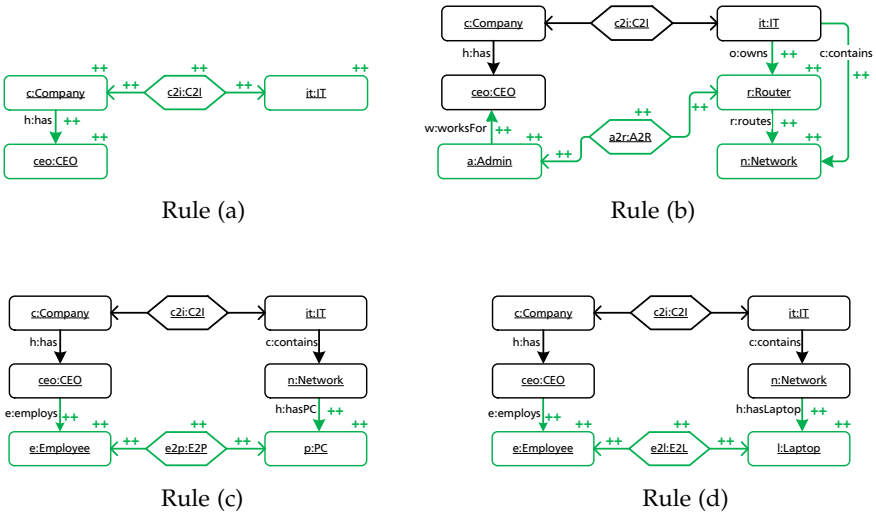
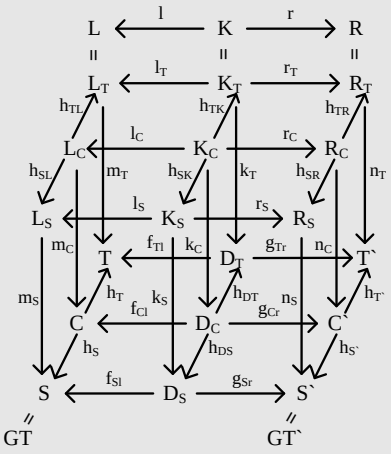


Figure 21: Complete set of TGG rules of our running example

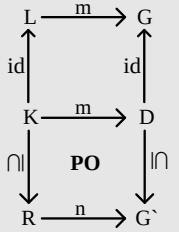
Definition 9 (Triple Graph Rule Applications)
Given a typed triple graph production rule $r = (L \xleftarrow{l} K \xrightarrow{r} R)$ *and a typed graph triple* $GT = (S \xleftarrow{h_S} C \xrightarrow{h_T} T)$ *with a typed graph triple morphism* $m : L \to GT$, *called the match, a direct typed graph transformation* $GT \overset{r@m}{\rightsquigarrow} GT'$ *from* GT *to a typed graph triple* GT' *is given by the pushout diagram depicted below.*



Remark: Obviously, the only difference between triple graph transformations and standard graph transformations is that all components of a rule and match are extended to triples. Hence, whenever possible we omit the complete drawing (i.e., extend every component to triples) and favor a short-hand notation.

To further improve readability, we define *monotonic creating rules* that reduce TGG rules to their necessary minimum:

Definition 10 (Monotonic Creating Rules)
A monotonic creating rule $r := (L = K, R)$*, is a pair of typed graph triples s.t.* $L \subseteq R$*. A rule* r rewrites *(via adding elements) a graph triple* G *into a graph triple* G' *via a match* $m : L \to G$*, denoted as* $G \overset{r@m}{\rightsquigarrow} G'$*, iff* $n : R \to G'$ *can be defined by building the* pushout G' *as denoted in the diagram.*



This definition shows, that a TGG rule is a condensed standard graph transformation rule, as L and K are identical.

We are now ready to define what a TGG consists of.

Definition 11 (Triple Graph Grammar)
A triple graph grammar $TGG := (TG, \mathcal{R})$ *consists of a type graph triple TG (also known as* TGG schema*) and a finite set* $\mathcal{R}$ *of monotonic creating rules. The generated language (*$G_\emptyset$ *denotes the empty graph triple) is* $\mathcal{L}(TGG) := \{G \mid \exists\, r_1, r_2, \ldots, r_n \in \mathcal{R} : G_\emptyset \overset{r_1@m_1}{\rightsquigarrow} G_1 \overset{r_2@m_2}{\rightsquigarrow} \ldots \overset{r_n@m_n}{\rightsquigarrow} G_n = G\}$.

Example

In our running example, the set $\mathcal{R}$ consists of the rules depicted in Fig. 21, and the TGG type graph depicted in Fig. 20.

Finally, we define consistency between related graphs:

Definition 12 (Consistent Integration) *Given two typed graph* $G_S \in \mathcal{L}(TG_S)$ *and* $G_T \in \mathcal{L}(TG_T)$*.* G_S *and* G_T *are* consistent *w.r.t. a given TGG iff* $\exists G_C \in \mathcal{L}(TG_C)$*:* $(G_S, G_C, G_T) \in \mathcal{L}(TGG)$*.*

Hence, consistency denotes the state where two integrated graphs fit properly together regarding a specific TGG.

Furthermore, a graph G_S (G_T) is *schema-compliant* if it is correctly typed over its type graph and, furthermore, could be derived with the source (target) domain component of the TGG rules.[3]

3 Note that schema-compliance is defined differently in cases where negative application conditions and other graph constraints are taken into account [AST12].

4 TGG CONTROL ALGORITHM CONCEPTS

So far, concepts of graph transformations, graph grammars and TGGs have been introduced. TGGs provide a declarative, rule-based means of specifying the consistency of source and target models in their respective domains, and tracking inter-domain relationships between model elements explicitly by automatically maintaining a correspondence model.

Although TGGs describe how *triples* consisting of source, correspondence, and target models are simultaneously derived, most practical software engineering scenarios require that source or target models already exist and that the models in the correspondence and the opposite domain are consistently constructed by a unidirectional forward or backward transformation. As a consequence, TGG tools that support bidirectional model transformation rely on unidirectional forward and backward operational rules, automatically derived from a single TGG specification as basic transformation steps, and use an *control algorithm* that decides which rule is to be applied on which part of the input graph.

Such an algorithm accepts a set of transformation rules and an actual graph to be transformed. To actually transform an input model, the algorithm has to fulfill two tasks:

(i) The algorithm has to find an appropriate traversal order through the input model.

(ii) To process a model element the algorithm has to choose the right rule in cases where more than one rule is applicable.

This chapter is organized as follows:

1. As TGG rules describe the evolution of all three domains at once, so-called *operational rules* have to be derived to allow for unidirectional forward or backward transformations. These operational rules will then be used by a control algorithm.
2. Variation points for TGG approaches and their control algorithm implementations are presented.
3. Formal properties of TGG control algorithms are discussed.
4. Concrete strategies of such algorithms are discussed.
5. The control algorithm of [KLKS10] is briefly reviewed as we will rely later on the rule selection capabilities of this approach.

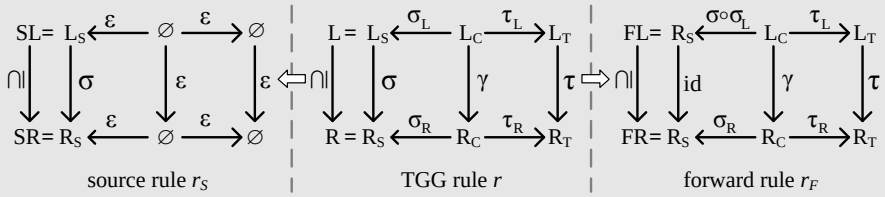
4.1 OPERATIONAL RULE DERIVATION

As we aim at transferring information from one model to another and, in addition, to propagate changes between models, we need again unidirectional transformation rules as TGG rules are not applicable. Such rules must be capable of accepting all possible input scenarios. In our context, such scenarios are for example to accept an existing model as input and create the appropriate opposite model or cope with the deletion of elements in one model and remove related opposite elements as well.

The declarative TGG rules describe the simultaneous evolution of all three domains, which means that these rules describe the set of consistent graphs. This knowledge is now used to derive so-called *operational rules* which are unidirectional. Hence, so-called *forward rules* are used to transfer data from the source to the correspondence and target domain, while so-called *backward rules* apply analogous changes in the opposite direction. The overall rule application process will be governed by a so-called control algorithm.

As shown in [Sch95, EEE+07], a sequence of TGG rules, which describes a simultaneous evolution, can be uniquely decomposed into (and conversely composed from) a sequence of source rules that only evolve the source model and forward rules that retain the source model and evolve the correspondence and target models.[1]

Definition 13 (Derived Operational Rules)
Given a $\mathsf{TGG} = (\mathsf{TG}, \mathcal{R})$ *and a rule* $r = (L, R) \in \mathcal{R}$*, a* source rule $r_S = (SL, SR)$ *and a* forward rule $r_F = (FL, FR)$ *can be derived according to the following diagram:*



Although forward rules retain all source elements, a control algorithm has to keep track of which source elements are transformed by a rule application. This can be done by introducing marking attributes [HEO+11], or maintaining a bookkeeping data structure in the control algorithm [GHL10]. In concrete syntax, we equip every

1 All arguments and definitions are symmetrically usable, which means although we always speak of forward transformations and forward rules, statements analogously hold for the backward direction.

transformed element with a *checked box*, and every *untransformed* element with an *unchecked box* as introduced in [KLKS10].

Example

From Rule (c) of our running example (Fig. 21), the operational rules r_S and r_F depicted in Fig. 22 can be derived. The source rule extends the source graph by adding an `Employee` to an existing `Company`, while the forward rule r_F *transforms* (denoted as $\square \rightarrow \boxtimes$) an existing `Employee` by *creating* a new `E2P` link and `PC` in the corresponding `Network`.

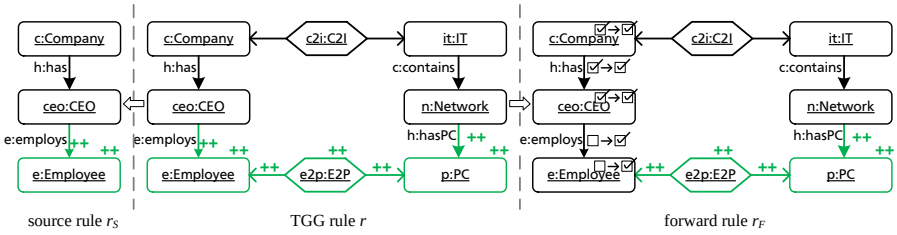


Figure 22: Source and forward rules derived from Rule (c) (cf. Fig. 21)

Regarding the forward rule, all elements in the source domain must exist. Hence, the additional checked and unchecked boxes depict, which elements will be transformed (i.e., $\square \rightarrow \boxtimes$) by applying this rule, or will be used as context (i.e., $\boxtimes \rightarrow \boxtimes$).

Additionally, it is also possible to derive other operational rules. Non-modifying rules, for example, may check whether an existing integration is consistent regarding the specified TGG. Another derivable operational rule could for example create traceability links between existing source and target models.

4.2 VARIATION POINTS OF TGG CONTROL ALGORITHMS

To understand the challenge of traversing an input model and selecting the appropriate rules, we discuss how existing algorithms handle the source graph of our example triple (Fig. 19) in a batch transformation. Processing an input graph requires certain decisions to be done by the control algorithm:

(i) First, the algorithm has to select an element to be processed and, if more than one element is available, choose an appropriate element.

(ii) Secondly, the algorithm has to decide which rule is to be applied in order transform the currently selected element. If more than one rule is applicable, the algorithm has to choose one.

In general, algorithms and their related TGG dialect can differ in three variation points: (i) properties of rules, (ii) strategies of finding an appropriate element traversal order, and (iii) strategies of selecting an appropriate rule for a single element.

(i) **Strategies of finding a model element traversal order**: This can be achieved in a bottom-up or a top-down manner.

Bottom-up strategies typically select one element to be processed randomly and process necessary context elements on demand. Hence, if the transformation of a certain element requires that another element was processed in advance, this element will be transformed on demand.

In contrast, top-down approaches try to find a processing order first (e.g., by analyzing the input model) and process the input in this specific order which guarantees that all preconditions for transforming a certain element are fulfilled.

(ii) **Rule properties**: We distinguish between *unrestricted*, *(local) confluent*, *functional*, and *local complete* TGGs.

Local confluence poses a restriction on TGG rules which requires that rules may not interfere with each other and, hence, may produce different results (i.e., TGGs must be globally deterministic). Nevertheless, local confluent TGGs cannot guarantee terminating transformation sequences.

Demanding functionality strengthens this restriction, as, additionally to local confluence, termination must be guaranteed (see Sect. 3.4.4 in [EEPT06]).

Orthogonally, local completeness ensures efficiency (no need for backtracking which results in an exponential worst case runtime complexity) for non-functional TGGs, i.e., non-determinism regarding possible results for the same input is allowed.

(iii) **Rule selection strategies**: One option is to completely prohibit a situation where more than one rule is applicable to the same model element. Another option is to require additional user input or to decide due to some externally provided heuristics. Nevertheless, considering such strategies is out-of scope for this thesis.

4.3 FORMAL PROPERTIES OF TGG CONTROL ALGORITHMS

Regardless of the chosen strategies and restrictions, transformation control algorithms should fulfill certain formal properties in order to produce reliable and trustworthy results. This directly reflects the requirement of verification from real-world application scenarios (cf. Chap. 2). Certain properties of general graph transformation systems

have been defined by Stevens in [Ste08b]. For example, every control algorithm should terminate its processing at a certain point. Otherwise, a transformation may last for such a long time, which cannot be distinguished from a transformation process that ran into an infinite loop.

For TGG control algorithms, Schürr and Klar stated in [SK08] three major properties that should be met by any TGG implementation: correctness, completeness, and efficiency. In the following, these properties are defined only for forward rule applications. All properties can be analogously defined for backward rule applications.

4.3.1 *Correctness*

Correctness is the quality of an algorithm to either terminate with a (meaningful) exception (e.g., if the input graph is not correctly typed) or to return a graph triple GT that complies with the defined language of the TGG (i.e., $GT \in \mathcal{L}(TGG)$).

Definition 14 (Correctness)
Given a source graph G_S*, the transformation algorithm is* correct *if it either terminates with an exception or produces a consistent graph triple* $G = G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$.

4.3.2 *Completeness*

Completeness means that for a given graph triple that has already been produced with the transformation algorithm it is possible to derive the same or another triple again, starting with the source or target graph as input. In other words: if a consistent graph triple $G = G_S \overset{h_S}{\leftarrow} G_C \overset{h_T}{\rightarrow} G_T$ exists and we start a batch or incremental transformation with G_S as input, the algorithm will find a consistent graph triple $G' = G_S \leftarrow G'_C \rightarrow G'_T$ or terminate with an appropriate exception if the given TGG and input violate some restrictions imposed by the selected control algorithm.

Definition 15 (Completeness)
For all triples $G_S \overset{h_S}{\leftarrow} G_C \overset{h_T}{\rightarrow} G_T \in \mathcal{L}(TGG)$*, the transformation algorithm is* complete *if it produces a consistent triple* $G' = G_S \leftarrow G'_C \rightarrow G'_T \in \mathcal{L}(TGG)$ *for the input source graph* G_S *or terminates with an exception.*

4.3.3 *Efficiency*

In our context, an algorithm is considered efficient, if its execution time is polynomial in the number of elements in the input graph or, in the incremental case, in the number of changes and affected elements. In the incremental case, additions are retained in the set Δ^+ while deletions are stored in the set Δ^-.

Definition 16 (Efficiency)
According to [SK08], a TGG batch transformation algorithm is efficient if its runtime complexity class is $\mathcal{O}(n^k)$, where n is the number of nodes in the source graph to be transformed and k is the largest number of elements to be matched by any rule r of the given TGG.
In the incremental case, the algorithm is efficient if the synchronization runtime effort is polynomial in the number of changes ($|\Delta^-| + |\Delta^+|$), their (potentially) dependent[a] and their direct context elements. This set of elements is denoted as n_δ. Hence, the incremental algorithm is efficient if it performs in the order of $\mathcal{O}(n_\delta^k)$.

[a] Note that different interpretations of dependent elements exist as already mentioned in Chap. 1.

From our point of view, a control algorithm should always be provable comply to all three formal properties. Hence, when introducing new algorithm approaches in Parts ii and iii we will always have to prove that these algorithms are correct, complete, and efficient.

4.4 BOTTOM-UP, CONTEXT-DRIVEN AND RECURSIVE

Regarding the variation points introduced in Sect. 4.2, we will discuss how concrete setups would look like.

An established strategy is to transform elements in a bottom-up context-driven manner, i.e., to start with a random node and check if all context nodes (dependencies) are already transformed *before* the selected initial node can be transformed. If a context node is not yet transformed, the algorithm transforms it, by recursively checking and transforming its context. Context-driven algorithms always start their transformation process with an arbitrarily selected node, without knowing if this was a good choice, i.e., if the node can be transformed immediately or if the input model as a whole is even incorrect. Such algorithms are correct, but, in general, have problems with completeness due to wrong *local* decisions as it cannot be guaranteed in general that it is the right choice to process a certain element right now and not to postpone its processing.

The term "bottom-up" refers to the strategy of deciding on demand which element will be processed.

4.4.1 *Unrestricted*

In order not to restrict the expressiveness of TGGs that can be handled by the transformation algorithm, a simple backtracking strategy could be employed to cope with wrong local decisions. Such a strategy tries to find a suitable transformation sequence to produce an appropriate result and, therefore, test in the worst-case all possible transformation sequences. If a backtracking algorithm returns no consistent graph triple, then it is clear that such a triple does not exist and, hence, the input graph is not part of the specified language.

Example

For our example, a first iteration over all nodes would determine that only `ES` together with `Andy` can be transformed by applying Rule (a). In a second iteration the algorithm would determine again in a trial and error manner that only `Ingo` can be transformed next with Rule (b), as neither `Tony` nor `Marius` can be transformed using Rule (c) or (d) (a `Network` is missing in the opposite domain). Finally, `Tony` and `Marius` can be transformed. This algorithm is correct and complete as shown in [EEE+07, Sch95] but has exponential worst-case runtime and is, therefore, impractical for real-world applications.

It is, however, possible to guarantee polynomial runtime of the context-driven recursion strategy by restricting the class of supported TGGs appropriately as in case of the following approaches.

A more complex but abstract backtracking scenario is depicted in Fig. 23. Assume that three elements e_1, e_2, e_3 have to be transformed. The algorithm starts with e_1 and is able to apply rule r_1. In a next step, element e_2 is chosen but applying rule r_2 fails. Hence, rule r_3, which is also applicable to element e_2 is successfully applied. Finally, element e_3 shall be transformed by applying rule r_4. As this fails and no other rule is applicable to e_3, the algorithm has to return to the transformation of e_2. As no additional rule is specified to transform e_2, the algorithm has to step back even further and reconsider the transformation of element e_2. Selecting e_3, the algorithm chooses to successfully apply rule r_4. As only e_2 remains untransformed, rules r_2 and r_3 are tested. As both applications fail and no other rules to transform e_2 and e_3 exist, the algorithm has to return to the transformation of element e_1. Here, another rule r_5 is applicable and, as a consequence, all other elements can be transformed successfully. Finally, the algorithm quits with a completely transformed model.

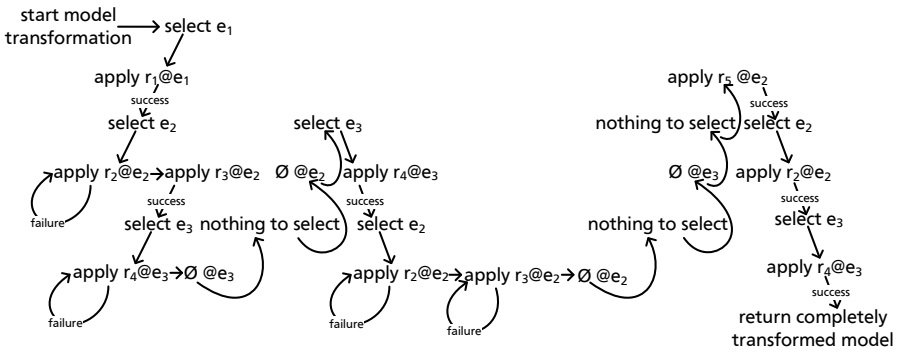


Figure 23: Complex backtracking scenario with a number of wrong choices

4.4.2 *Functional Behavior*

Demanding *functional behavior* [GHL10, HEGO10] guarantees that the algorithm can choose freely between applicable rules at every decision point and will always get the *same result* without backtracking. Functional behavior, therefore, restricts a TGG to be a function over graph triples, where for a given graph always the same result is returned, regardless how often the transformation is repeated with the same input. The result can either be a consistent graph triple, or an exception as the input graph was not part of the specified language (cf. Def. 12). Figure 24 depicts this process graphically. Regarding a transformation process, the algorithm may have the same choices as before (decide which element will be processed and which rule will be applied). Depending on this decision, different intermediate states are possible. These intermediate states differ in which elements are already processed or which rules have been applied. Altogether, the transformation either ends up in an erroneous situation, as the local decision did not lead to a valid state, or leads to the same resulting graph triple.

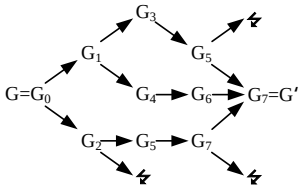


Figure 24: Schematic overview of a transformation process with functional TGGs

Such a TGG also requires functional behavior of the backward rule application. Hence, functional TGGs specify a bijective function. Although functional behavior might be suitable for fully automatic integrations, our experience with industrial partners shows that user interaction or similar guidance (e.g., configuration files) of the in-

tegration process is required and leads naturally to non-functional sets of rules with certain degrees of freedom [Kö5, LSRS10, RLSS11]. Please note that our running example is clearly non-functional due to Rules (c) and (d), which can be applied to the same elements on the source side, but create different elements on the target side. Therefore, depending on the choice of rule applications, *different* target graphs are possible with our running example. Demanding functional behavior is a strong restriction that reduces the expressiveness and suitability of TGGs for real-world applications [SK08, KLKS10]. Nevertheless, such a strategy has polynomial runtime and its applicability can be enforced statically via critical pair analysis [EEPT06]. Termination for a set of TGG rules can be guaranteed as soon as every rule contains at least one element in every domain that is in $R \setminus L$ (i.e., created element) [HEOG10].

4.4.3 *Local Completeness*

Algorithms that allow a non-functional set of rules to handle a larger set of scenarios have to exploit additional implicitly modeled information of the transformation to cope with non-determinism and non-bijectivity [Ste08b], while still guaranteeing completeness for a certain class of TGGs. Hence, [KLKS10] demands *local completeness*, i.e., that a local decision between rules that can transform a certain node *cannot* lead to a dead-end. This means that a local choice (which can be influenced by the user or some other means) might actually result in *different* output graphs, which are, however, always consistent, i.e., in the defined language of the TGG ($\mathcal{L}(TGG)$).

Example

For our running example (cf. Figs. 15 and 21), we could start with an arbitrary node, e.g., `Ingo`. According to Rule (b), a `CEO` and a `Company` are required as context and Rule (a) will thus be applied to `ES` and `Andy`. After processing `Ingo`, `Tony` and `Marius` can be transformed in an arbitrary order, each time making a local choice if a `PC` (Rule (c)) or `Laptop` (Rule (d)) is to be created. Note that our running example is *not* local complete, as it cannot be decided whether an `Admin` or an `Employee` should be transformed first (Rules (c) and (d) demand an element on the *target* side that can only be created by Rule (b)). For this reason, the algorithm might fail if it decides to start with one of the `Employees`. In this case, Rules (c) and (d) would state that `ES` and `Andy` are required as context and have to be transformed first. This is, however, insufficient as a `Network` must be present in the target domain as well. This context-driven approach fails here as transforming `ES` and `Andy` with Rule (a) *does not* guarantee that the employees `Marius` and `Tony` can be transformed. The problem here is that context-driven algorithms only regard the given input graph for controlling the rule

application and do not consider *cross-domain context dependencies* such as `Network` in this case.

4.5 TOP-DOWN AND ITERATIVE

In contrast to context-driven recursive strategies, which lack a *global view* on the overall dependencies and seem to be unsuitable for an *incremental* synchronization scenario, algorithms can operate in a top-down iterative manner exploiting a certain global view on the whole input graph instead of arbitrarily choosing a node to be transformed.

The term "top-down" refers to the strategy of employing additional knowledge regarding the models under consideration. Such additional knowledge is for example a topological sorting of the input model, which reveals dependencies and, therefore, a basis for determining the traversal order.

4.5.1 *Unrestricted*

The algorithm presented by [KRW04] requires that all TGG rules demand and create at least one correspondence link, i.e., a hierarchy of correspondence links must be built up during the transformation. The correspondence model can be used to store dependencies between links in this case and is interpreted as a directed acyclic graph, which is used to drive and control the transformation. This algorithm is both batch *and* incremental but it is unclear from [KRW04] for which class of TGGs completeness can be ensured. The most recent implementation [GR11] seems to utilize backtracking and, hence, it can be concluded that this approach fulfills completeness and correctness but suffers from inefficiency.

4.5.2 *Functional Behavior*

Another approach that utilizes a global view is presented by Giese et al. in [GHL10]. This approach demands again for functional behavior but it can efficiently process models. Similarly to the previous approach, they also utilize an exhaustive correspondence data structure, that allows for building a directed acyclic graph. For the batch transformation case, this approach is proved to be complete and correct. It is to be expected that this is also the case for the incremental case, but these proofs are not yet presented.

4.5.3 *Local Completeness*

Restricting TGGs to comply to local completeness, allows for a precedence-driven strategy. Such a strategy defines and uses a partial order of nodes in the source graph according to their *precedence*, i.e., the sorting guarantees that the nodes can only be transformed in a sequence that is compatible with the partial order. This approach is novel in the domain of non-functional TGGs and seems to be a promising means of coping with incremental changes efficiently, while retaining user added information as much as possible. Upcoming parts of this thesis will deal with the introduction of this approach.

4.6 THE KLAR CONTROL ALGORITHM

The starting point for the concepts presented in this thesis, was the algorithm implemented in MOFLON [AKRS06, KRS09]. This algorithm was a context-driven batch algorithm, which additionally provided features such as priorities and rule inheritance.[2] The formalization of this algorithm has been presented by Klar et al. [KLKS10] where additionally negative application conditions have been introduced in favor of priorities and rule inheritance. This algorithm has been finally implemented in our tool eMoflon [ALPS11, ALS12].[3]

It is necessary to review this algorithm, as the introduction of our new control algorithm reuses a very specific part: As stated in the introduction, our purpose in this thesis is to develop an approach that efficiently propagates model changes and retains information. Therefore, we research how to determine an appropriate traversal order, which is one of the two tasks of a control algorithm. The other task, selecting an appropriate rule to process an element, was out-of-scope and, hence, this part will be reused from the here presented algorithm.

Short Review

This section, reviews briefly the TGG batch algorithm of [KLKS10] and its basic concepts on an abstract level. The complete formalization is presented in [KLKS10] and the PhD thesis of Felix Klar [Kla12].

Algorithm O[4] works in different stages. First, the algorithm arbitrarily selects a node to be transformed (lines 2–4). If this node has not

2 Abilities regarding rule inheritance have been investigated in [WKK+11, WKK+12].
3 Download eMoflon from http://www.emoflon.org/
4 This algorithm is referred to as "Algorithm O" as it is not a contribution of this thesis.

Algorithm O Batch Context-Driven Control Algorithm

```
1: procedure TRANSFORM(G = G_S ←^{h_S} G_C →^{h_T} G_T)
2:   for all nodes n ∈ G_S do
3:     TRANSFORM(n)
4:   end for
5: end procedure
6: procedure TRANSFORM(node n)
7:   if n has been visited previously then
8:     throw cycle exception
9:   else
10:    candidateRules ← rules that could transform n
11:    appropriateRules, applicableRules ← ∅
12:    for all rules r ∈ candidateRules do
13:      check dangling edge condition (DEC) for applying r@n
14:      if DEC is satisfied then
15:        if all context elements c are transformed then
16:          appropriateRules ← appropriateRules ⋃ r
17:        else
18:          for all untransformed context nodes c do
19:            TRANSFORM(c)                ▷ Recursively transform all nodes
20:          end for
21:        end if
22:      else
23:        continue                                          ▷ Skip this rule
24:      end if
25:    end for
26:    for all rules r ∈ appropriateRules do
27:      check if DEC for applying r@n                ▷ DEC could be violated
                                          due to problematic context transformation
28:      if DEC is satisfied then
29:        nIsLocallyTransformable = true
30:      end if
31:      if the match can be completed in all other domains then
32:        applicableRules ← applicableRules ⋃ r
33:      end if
34:    end for
35:    if applicableRules = ∅ then
36:      if nIsLocallyTransformable then
37:        throw local completeness violated exception
38:      else
39:        throw G ∉ ℒ(TGG) exception
40:      end if
41:    else
42:      if |applicableRules| ⩾ 1 && ∀r ∈ applicableRules : transform same
   set of nodes then
43:        select one rule r ∈ applicableRules and apply r@n
44:      else
45:        throw competing core match exception
46:      end if
47:    end if
48:  end if
49: end procedure
```

been regarded previously, the transformation process may go ahead, otherwise we ran into a cycle (lines 7–9) and an exception is thrown.

The next stage is to collect all rules that may be used to transform the actual node n (lines 10–23). In this phase the so-called *dangling edge condition* (DEC) is checked for the very first time (line 13).

Informal Definition[5] (Dangling Edge Condition):
The dangling edge condition is a formal description of the fact that after having applied a certain rule to node n all untransformed incident edges are still transformable by applying other rules of the TGG.

If DEC is satisfied, then the algorithm recursively tries to process all context nodes (line 16) and if all context nodes have been transformed then this rule is added to another set (line 18–20).

Having transformed all context nodes successfully, the algorithm enters the next stage, in which the potentially applicable rules are filtered further: A re-check of the DEC determines if any of the recursive context transformations changed the situation that was found in line 13. If so, the algorithm has definitely found a rule that can transform n at least in G_S (line 27). This rule is applicable if its match can also be completed in all other domains (lines 29–31).

Finally, the last stage (lines 33–45) is to actually apply a rule to transform n. For this reason a rule is selected and applied in line 40. All other statements in this stage deal with detecting problems within the specification of the TGG (e.g., local completeness violation (line 35)), finding incorrect input triples (line 37), or a combination of both (line 43).

4.6.1 *Restriction to Local Complete TGGs*

As described shortly in the algorithm comparison of this chapter, until now the introduction of *local completeness* seems to be a sufficient restriction of TGG specifications in order to allow for non-functional TGGs (i.e., allowing for a set of resulting consistent graph triples) and to avoid backtracking for efficiency reasons.

Informal Definition[6] (Local Completeness Criterion):
A TGG is local complete iff applying the source rule r_{i_S} of a TGG rule r_i successfully guarantees that there exists a forward rule r_{j_F} (i may be equal to j) that can also be successfully applied.

This criterion is of immense importance to us as it allows us to *consider one domain only* in order to find an appropriate transformation

5 For this thesis, only an informal definition is needed to understand the concept of dangling edges. The formal definition is presented in [KLKS10].

6 Again, only an informal definition of this concept is needed to understand all coming parts and, therefore, I decided to present an informal definition only. The formal definition can be found in [KLKS10].

sequence and, therefore, transformation result. Unfortunately, up to now it is impossible to statically check (at compile-time) if a TGG specification fulfills the local completeness criterion. This is still an open research topic.

Note that the approach presented in this thesis works with local complete TGGs only. Hence, the running example will be adjusted to fulfill this property in Sect. 6.

4.6.2 *Open Questions*

The algorithm presented above has the following drawbacks:

- The recursive transformation process may run into cycles which are detected and stop the transformation process with an exception. It is then unclear whether the exception has been triggered due to the fact that the TGG is not compatible with the selected transformation algorithm or whether the input graph is not correctly typed and/or cannot be transformed by the given TGG. Furthermore, the cycle detection may occur at the very end of an expensive transformation process that should not have been started anyway.
- Keeping track of just regarded elements of the input graph, the corresponding rules and their matches requires an expensive bookkeeping effort. Hence, recursive transformation stacks may become large depending on the size of the input graph.
- Up to now, it seems that context-driven strategies are not suitable for efficiently determining the affected elements due to an incremental change. This is important as incremental changes may lead to incorrect (sub-)graph triples, whose transformation has to be revoked before starting a new transformation process.

5 RELATED TOOLS AND OTHER TGG APPROACHES

This chapter provides an overview on model transformation approaches that are in use and/or under development. Some of them are academic prototypes (e.g., AToM^3, Henshin, GRoundTram, MOFLON or eMoflon), while others are used in practical scenarios due to their maturity (e.g., ATL and ETL). Nevertheless, all of them have their very own list of featured properties.

A second aim of this chapter is to discuss how other existing TGG approaches could be used to achieve our goal of efficient and information preserving model synchronization. Hence, the part related to TGG approaches is more detailed compared to the general introduction of other transformation approaches.

5.1 GENERAL OVERVIEW

As model transformation is a challenge in various application domains and communities, there exists a substantial number of different concepts, languages, and tools [CH06, Ste08a]. Basically, we distinguish between unidirectional and bidirectional model transformations.[1]

Regarding the general benefits and drawbacks of unidirectional vs. bidirectional model transformations, the following has to be considered:

- Unidirectional model transformations benefit from a clear information flow within the actual transformation specification and, hence, it feels more natural to specify such a transformation compared to a bidirectional one.
- A direct drawback is that when bidirectionality is a must, two unidirectional transformations have to be specified (i.e., one forward and one backward transformation) in contrast to a single bidirectional specification.
- Specifying two unidirectional model transformations consistently (i.e., without contradictions) can be a complex task. This is especially true when maintaining such a pair of unidirectional transformations requires lots of experience to always consider all

1 Additionally, multidirectional model transformations are also possible and defined [OMG08]. In practice, such transformations can be decomposed into bidirectional transformations and are, therefore, not considered here.

facets of the transformation pair completely. In the bidirectional case, the person that specified the transformation is no longer responsible for guaranteeing the consistency between derived or interpreted forward and backward transformations as this is an implicit feature of the bidirectional transformation approach itself.

- Finally, expressiveness regarding the set of possible features within a model transformation (e.g., control flow components, complex attribute handling, etc.) differ. Typically, unidirectional approaches are more expressive as each language feature needs to operate in a single direction only. Hence, in some scenarios (e.g., with complex calculations that do not allow for an automatic reverse use) bidirectional transformation approaches may be unsuitable.

As [Ste08a] and [CH06] provide a comprehensive overview and categorization of model transformation approaches, we only survey a small subset of existing techniques. The selection here is based (i) on the subjectively noticed proliferation of an approach and (ii) on the relation compared to the approach presented in this thesis. The latter criteria explains why numerous TGG approaches have been reviewed.

5.2 UNIDIRECTIONAL MODEL TRANSFORMATION APPROACHES

Since unidirectional approaches are distinguished by performing a transformation *inplace* or *outplace*, we consider representative frameworks of both fields. Inplace model transformations use one model as input and modify exactly this model in order to produce an appropriate output model. Hence, the input model has been replaced after applying a transformation. Selected unidirectional inplace approaches are *A Tool for Multi-formalism and Meta-Modelling* (AToM^3) [dLV02a] and *From UML to Java and Back Again* (Fujaba) [KNNZ99] with its transformation approach Story Driven Modeling (SDM) [FNTZ00]. In contrast, outplace model transformations accept one input model and produce an appropriate output model without modifying the input. Exemplary unidirectional outplace approaches are *ATLAS Transformation Language* (ATL) [JABK08] and *Epsilon Transformation Language* (ETL) [KPP08].

Considering the requirements of Chap. 2, unidirectional transformations are not sufficient to cope with a complex real-world scenario such as our motivational example.

5.2.1 *Inplace Transformation Approaches*

A Tool for Multi-formalism and Meta-Modelling

A Tool for Multi-formalism and Meta-Modelling (AToM^3) is a CASE-tool[2] [dLV02a, dLV02b]. It is implemented in Python and aims at modeling data structures as well as model transformations. The basic idea is that every data structure can be specified as typed graphs. The formalism used for model transformations is a triple graph replacement system. Specifications can be modeled in a hybrid manner meaning that graphical and declarative graph patterns can be extended with Python method calls in order to equip transformation designers with imperative techniques. AToM^3 supports incremental model transformations with a sophisticated event-driven concept [GdL04] that claims not to be bound to a certain editor in contrast to syntax-driven incremental approaches. In order to support modularization the AToM^3 rewrite engine allows for sequential execution of multiple graph grammars that may belong to different tasks (e.g., first transform a model with grammar `a`,. then optimize the model with grammar `b`). Additionally, transformation designers are allowed to use the concept of negative application conditions (NACs) with AToM^3 [TEG$^+$05] to specify constraints when a rule should not be applied. Unfortunately, AToM^3 seems to be no longer under development and is additionally coupled to its own data model. De-facto modeling standards such as EMF/Ecore are not supported.

Story Driven Modeling

Story Driven Modeling (SDM) [FNTZ00, vDRH$^+$11] is a composition of declarative graph transformations and imperative control flow definitions. It is therefore possible to define behavior of a method via expressing model modifications with unidirectional graph transformation rules and additionally embed these rules into control flow elements such as statements, loops and conditions. SDM is implemented in the tool Fujaba [KNNZ99] which utilizes this model transformation approach to define the behavior of method implementations. SDM specifications are both, declarative and imperative: Next to describing model modifications with graph patterns (declarative), also a fallback solution to pure Java code can be used to specify actions within a control flow (imperative).

2 Computer Aided Software Engineering

5.2.2 *Outplace Transformation Approaches*

Atlas Transformation Language

The *Atlas Transformation Language* (ATL) [JK06, JABK08] is a unidirectional textual transformation language. Standard ATL does not support traceability between source and target models as a feature, but this can be achieved by using an extension called AmmA [DV07], or by explicitly adding trace elements and their evolvement to the rules. An ATL implementation is available as an Eclipse plugin which supports the de-facto standard modeling language Ecore. Furthermore, precise semantics for ATL 3.0 have been introduced and proved by Troya and Vallecillo in [TV11] by using formal logic.

Epsilon Transformation Language

The *Epsilon Transformation Language* (ETL) [KPP08, KRPGD11] is one out of many task specific languages that are built upon the Epsilon Object Language (EOL) within the Epsilon framework. 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 was furthermore designed to overcome the restrictions of OCL but still adhere to precise semantics. ETL was developed with the focus on model transformations. ETL is a hybrid approach allowing for declarative patterns and an imperative fallback to call EOL methods. Traceability is implicitly maintained. This allows for additional cross model checking with the Epsilon Verification Language (EVL). The model transformation engine can be run not only in a batch manner but also (semi-)interactively by specifying rules with additional user input only. Incremental model transformations utilize the automatically established implicit trace model. Originally, ETL is designed as a unidirectional transformation language but can be extended with EVL definitions to allow for bidirectional transformations.[3] ETL is reusable due to its precise semantics of inheritance [WKK+11, WKK+12] and modules (via EOL modules). Inheritance allows for the specialization of rules and EOL modules allow for the reuse in different transformation specification. Furthermore, ETL is not limited to transformation specifications that only cope with one source and target model, but with m source and n target models simultaneously. Unfortunately, ETL is not based on formal semantics [KRPGD11] and, hence, cannot be verified.

3 Source: Informal discussion with Richard F. Paige in January 2011 in Dagstuhl [HSST11].

5.3 BIDIRECTIONAL MODEL TRANSFORMATION APPROACHES

Regarding bidirectional approaches it is typical, that model transformations modify models outplace. Hence, other criteria are considered to distinguish them. *Algebraic* approaches are based on a formal algebraic foundation that defines the properties. Exemplary algebraic approaches are GRoundTram [HHKN09] and lenses [FGM+05].

Both bidirectional transformation approaches can be used to incrementally synchronize models. Unfortunately, they lack an explicit traceability model, which allows users to specify relationships between models.

5.3.1 *Algebraic Transformation Approaches*

GRoundTram

GRoundTram [HHKN09, HHI+10, HHI+11a, HHI+11b] is a bidirectional and hybrid graph transformation approach. The underlying formalism is the graph algebra UnCAL. Based upon UnCAL the graph query languages UnQL and its extension UnQL+ (with additional compositional features) are defined. Bidirectional model transformation in the context of GRoundTram demands for some additional explanation: With GroundTram, models can be transformed from a source model into a target model. Changes within the target model can be incrementally propagated back to the source model via the implicit trace model, which is hidden from the user. Unfortunately, it is not possible to start with a target model and create the appropriate source model from scratch. This is due to the fact that transformation specifications written in UnQL+ are unidirectional only and incremental backward transformation is achieved via the trace model only. Together with metamodel definitions in KM3, graphs can be queried for specific patterns and, furthermore, target graphs can be created and maintained. Traceability is an implicit feature as it is used for incremental updates and backward transformations but cannot be defined explicitly (i.e., by a correspondence model). In addition to a well-formalized and sound definition of underlying technologies there exists a reference implementation that allows for textual transformation definitions and graphical concrete model transformations.

Lenses

The lenses have been introduced by Foster et al. in [FGM+05]. Lenses denote an algebraic approach to specify views for models. Therefore, different laws have been postulated that ensure properties such as consistency between a model and its view while both may evolve over

time. A lens is basically a pair of transformations: a get, for the source to view transformation, and a put, for view to source. In practice, different implementations exist that support for example string manipulation [BFP+08, FPP08]. Furthermore, TGGs have been shown to be a concrete implementation of the symmetric lens framework [DXC+11].

Query/View/Transformation

The OMG specification *Query/View/Transformation* (QVT) [OMG08] defines a standard for model transformation on MOF compliant models. The specification provides the declarative language *QVT Relations* for bidirectional model transformations. Besides this, an additional unidirectional and imperative transformation language named QVT Operational Mappings, is also defined in [OMG08]. Considering QVT Relations, the major deficiency is that its specification is imprecise [Ste08b]. As a consequence, different tools claim to implement the QVT standard, but behave differently and incompatibly [Gua09]. Nevertheless, ongoing activities include implementing the Relations language as an Eclipse plugin (mediniQVT [ikv12]).

5.3.2 *Triple Graph Grammar Approaches*

Amongst the numerous bidirectional model transformation approaches surveyed in [Ste08a], the concept of TGGs features not only solid formal foundations [Sch95, EEE+07, KLKS10] but also various tool implementations [KRW04, GH09, GHL10, KLKS10, GR11, ALPS11, ALS12].

The TGG approach is based on category theory and obviously supports traceability (cf. Chap. 3). Furthermore, incrementality and efficiency are recently researched as well as possibilities to prove specific properties and, hence, distinct qualities regarding the expected results of a model transformation (cf. Sect. 4.3). As TGGs provide traceable and bidirectional model transformations, it seems to be a natural choice to select a TGG implementation for a scenario such as described in Chap. 2. Furthermore, properties of TGGs can be verified, while recent implementations show that also practical validation is possible.

In the following, we review the TGG approaches of Ehrig et al. (TGG_1) [EEH08], Kindler et al. (TGG_2) [KRW04] and Giese et al. (TGG_3) [GW06]. In addition, MOFLON [AKRS06, KKS07], the ancestor of our tool eMoflon [ALPS11], will also be reviewed briefly.

TGGs have been formally introduced in Chap. 3. In addition, this section is intended to discuss which differences between different TGG approaches exist and if it was possible to achieve the same aims, information preserving and efficient model synchronization,

with other TGG approaches. Furthermore, we briefly overview the strengths and weaknesses of these approaches.

TGG_1: Ehrig et al.

Providing formal aspects for incremental updates that guarantee well-behavedness according to a set of laws or properties is quite a challenge. Algebraic approaches such as lenses [DXC+11] and the framework introduced by Stevens [Ste08c] provide a solid basis for formalizing concrete implementations that support incremental model synchronization.

The TGG approach developed at the working group of Hartmut Ehrig [GEH10] (TGG_1) started as a fully-fledged formalization of TGGs. Inspired by [DXC+11], a TGG model synchronization framework was presented in [HEO+11, HEEO11] that is correct and complete. The proposed incremental algorithm, however, requires a complete re-marking of the entire graph triple and, therefore, it depends on the size of the related graphs and not on the size of the update and affected elements. This is infeasible for an *efficient* implementation and the need for an improved strategy has already been stated as future work in [HEO+11]. Furthermore, the re-marking process is not able to retain manually added information, while additionally non-determinism on the rule level (i.e., non-functional) is prohibited.

Their focus is to prove certain aspects of graph transformations and therefore rely on the widely examined Algebraic Graph Grammar (AGG) framework and a Mathematica implementation for mathematical proofs. Properties such as user-friendly rule specifications or efficient model transformation seem not to be a primary focus of the developers. Recent developments are changing this impression as members of this group started implementing a TGG tool named HENSHINTGG [EHGB12] that utilizes the unidirectional graph transformation tool HENSHIN for EMF/Ecore in Eclipse.

TGG_2: Kindler et al.

The TGG interpreter described in [GPR11] employs an transformation approach that focuses on introducing new features and favors usability over formality. Unfortunately, properties, such as efficiency, correctness, and completeness, cannot be guaranteed formally as stated in [GR11, GPR11].

The strength of this approach is that it tries to *reuse* elements instead of deleting and creating them. This is important as it prevents information loss that cannot be recovered by (re-)creating an element (user added contents) and is, therefore, in one line with our aims at retaining as most as possible user added information.

TGG_3: Giese et al.

Another incremental TGG transformation algorithm (based on previous work of Wagner et al. [GW06]) has been presented in [GW08, GH09], which exploits the correspondence model to determine an efficient update strategy. Therefore, this algorithm calculates the set of actually affected elements of a change as close as possible compared to the set of potentially affected elements (cf. Chap. 1). As this approach is already efficient, it seems on a first glance that this approach is a suitable base to achieve the goals of this thesis.

Unfortunately, the approach requires a strict binding to Eclipse in order to get notified about model changes. The drawback is that in practical scenarios it cannot be required that engineers maintain their models to be synchronized in Eclipse exclusively. Hence, a model difference recognition mechanism has to be employed that determines model changes by comparing two models (e.g., version i and $i+1$ of a specific model). Unfortunately, such a model diff consumes additional runtime that has to be added to the actual transformation runtime. In practice such diffs can be achieved for example on an XML basis quite efficiently in an $O(n_i + n_{i+1})$ complexity [WDC03], where n denotes the number of elements in a model that have to be compared. As model transformations with TGGs can be executed with an $O(n^k)$ runtime complexity (cf. Sect. 4.3), the overall process still complies to the complexity of $O(n^k)$.

Thus, utilizing a notification framework can become a source of additional and unnecessary model synchronization runs: Consider a model evolves and a certain element is created, used and deleted. Using a notification framework at least two synchronization runs have to be applied, which are propagating the creation and deleting the element. In contrast when using a model diff, no change would be detected and, hence, no synchronization effort is necessary. Assuming that engineers do not produce the final product at once, but use intermediate model states, it seems to be a legal argument to prefer a (costly) model diff after a complex model modification has been accomplished, instead of propagating intermediate, temporary, or even false model changes that have to be revoked later by applying even more synchronization runs.

Another drawback of the approach of Giese et al. is that they require functional behavior. This avoids non-determinism regarding rule applications but also restricts the class of TGGs as explained in Chap. 4.

Although the batch mode of this algorithm has been formally presented in [GHL10], the incremental version has not been fully formalized and it is unclear how the update propagation order is determined for changes to elements that are not linked via the correspondence model to other elements.

Both approaches ($TGG_{2,3}$) use an interpreter based implementation that interprets TGG rules in the case of [GPR11] or pre-compiled operational rules in the case of [GH09]. Unfortunately, both approaches lack the option to employ their transformations on other platforms than Eclipse as both are implementations are Eclipse plugins.

Altogether, approaches $TGG_{2,3}$ support incremental model transformation with information preserving capabilities already, but pose other restrictions that cannot be overcome easily to fit to a real-world scenario such as described in Chap. 2.

MOFLON

MOFLON [AKRS06, KKS07] was another tool that also implemented TGGs. Explicit traceability links have been maintained at run-time and, together with a tool integration environment (TiE) [KRS09], resulting graph triples are visible in an easy-to-use and comfortable manner. Additionally, MOFLON supported incremental model transformation already but only on a low level (inefficient and partially unpredictable) without formalized concepts.

A crucial fact about MOFLON was that the TGG implementation has not been formally founded but was used to introduce additional concepts, such as rule priorities, or rule inheritance. This lead to a complex and hardly maintainable implementation that was based on a deprecated Java standard (i.e. JMI) which is not compatible to the de-facto standard of Ecore models. Hence, MOFLON evolved in 2010/2011 to *eMoflon*.

eMoflon

eMoflon [ALPS11, ALS12] is the successor of MOFLON. Most concepts were ported to the de-facto modeling standard EMF/Ecore and, finally, eMoflon builds a bridge for exchanging model data with various other EMF/Ecore based tools. The implemented batch transformation algorithm was formalized in [KLKS10] and [Kla12]. The underlying model transformation technology for deriving operational rules is based upon Story-Driven Modeling [FNTZ00] and will be reviewed in detail in Chap. 18. eMoflon is the first tool that implements the efficient and formally founded batch algorithm introduced by Klar et al. in [KLKS10]. Unfortunately, this algorithm supports batch transformations only and, hence, needs further improvements to be able to transform models incrementally with information preserving capabilities.

Part II

BIDIRECTIONAL MODEL TRANSFORMATION WITH TGGS

The aim of this thesis is to present a novel way of synchronizing models with TGGs efficiently and with information preserving capabilities. Therefore, based on our contributions [LAVS12a, LAVS12b], we propose in the following chapters:

(i) a so-called *precedence analysis*, which statically analyzes a TGG specification and collects relevant information to determine an appropriate traversal order for a given input model at runtime (see Chap. 6).

(ii) a novel control algorithm for batch transformations that uses the information derived by the precedence analysis to define a model element traversal order (Chap. 7). Thus, this algorithm provably complies to all formal properties.

As mentioned before, the presented algorithms base on each other and reuse the rule selection capabilities of the Klar algorithm presented in Sect. 4.6. Figure 25 depicts how the different algorithms are built upon each other, where the grey bordered box means that this component will not be regarded in this part. The reused part of the Klar algorithm is depicted in a shaded box to reflect that this component will be treated as a black-box. Thus, this reused part denotes the rule selection component of our new control algorithm.

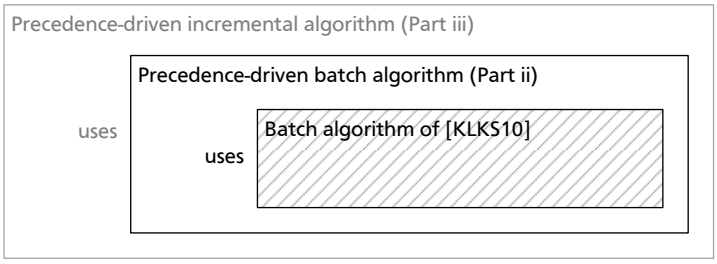


Figure 25: Schematic overview of building the batch (and incremental) algorithm

6

PRECEDENCE ANALYSIS

The idea of *precedences* is to express that an element y has to be transformed *after* another element x. This knowledge can be retrieved from the declarative TGG rule specifications and will further be used to define a partial order for the input model. This partial order will then be used to traverse the input model without any need for backtracking in a top-down manner as an appropriate starting point is known in advance. The challenging task is to determine such a (partial) transformation order in a way that it is guaranteed that this order is actually applicable.

The approach of processing data in a specific order based on precedences has been presented first for simple string grammars in the late 1960s by Wirth and Weber [WW66]. The idea here was to parse text efficiently in a bottom-up manner and to predict when to shift or reduce elements while parsing [ASU86]. This idea was lifted in the late 1970s and 1980s by various researchers to context-free graph grammars [Fra76, Fra78, Nag79a, Kau83, Kau86, Kau87]. Here, precedences are used to allow for efficient graph parsing as long as the graph grammar under consideration has the following three properties: (1) being context-free, (2) being confluent, and (3) being uniquely invertible (i.e., symmetric) [Nag79a, Kau86]. With these properties at hand, a so-called precedence table can be derived at compile-time from the rules of a grammar and is used to determine local precedence maxima ready for reduction.

All these different approaches of precedence grammars analyze the rule specification of a grammar at compile-time and derive certain information used at runtime to efficiently determine a traversing order of a given instance of the grammar (i.e., text or graph). As precedence graph grammars require confluence (i.e., all variations within a derivation process for a concrete instance lead to the same result) and context-freedom, we cannot utilize these approaches directly for TGGs, because TGGs are context-sensitive[1] by definition and confluence is a restriction we do not want to impose (cf. Chap. 4). Precedence graph grammars, therefore, are an inspiration only to build a TGG dialect that utilizes compile-time checks to define at runtime a

1 As TGGs do not support the deletion of elements in triple rules, non-terminals cannot be removed and, hence, TGGs have to be context-sensitive. As shown in [Nag79b], this is no general drawback, as context-free and context-sensitive graph grammars have the same expression power.

(partial) order for the input graph by considering this graph only (i.e., G_S in case of a forward and G_T in case of a backward transformation).

The overall process depicted in Fig. 26 shall work as follows:

I Specify a TGG with a TGG schema (i.e., type graph triple) and a set of rules.

II Analyze the rule specifications and build a so-called *precedence function* expressing the precedence relationship between distinct types.

III The input graph is traversed at runtime and a so-called *precedence graph* will be calculated by using the previously calculated precedence function. This precedence graph reflects all precedences of the input graph.

IV Transform the input graph according to the partial order induced by the precedence graph.

V After transforming an actual element of the input graph, update the precedence graph and repeat from step IV until all elements are transformed.

Hence, in the following sections, an analysis is presented that processes TGG rule specifications to retrieve information that can be used later by the precedence-driven control algorithm to a determine an appropriate traversal order.

Restriction: In order to be able to reuse the rule selection and application component of the control algorithm of [KLKS10] (cf. Sect. 4.6), we restrict also this approach to local complete TGGs. This is necessary to be able to prove formal properties such as completeness, while allowing for non-functionality.

Example

The rules presented in Sect. 3.3 do not form a local complete TGG: Rules (c) and (d) demand as context a `Network` object in the target domain in order to apply one of these rules. Unfortunately, this object has no equivalent in the source domain and for this reason it cannot be guaranteed that Rules (c) or (d) are already applicable in a forward transformation by considering the source domain only. Hence, the local completeness criterion is violated (cf. Sect. 4.6.1).

For this reason, Rules (c) and (d) are adjusted such that an additional `Admin` object in the source domain, is connected to the additional `Router` object in the target domain. This slight adjustment guarantees local completeness. Altogether, the rules depicted in Fig. 27 build our TGG.

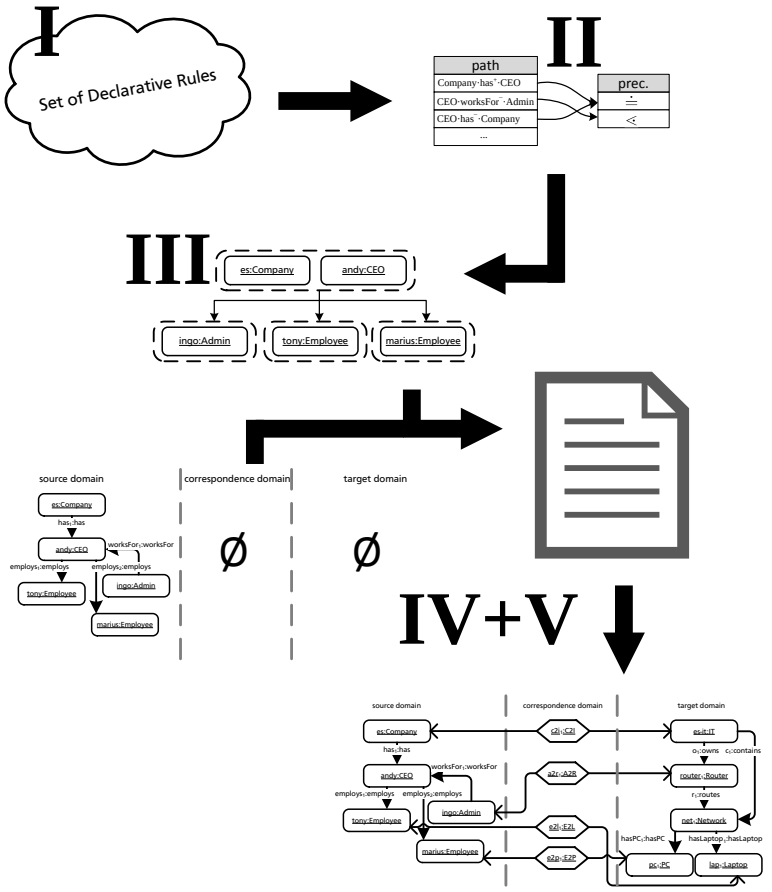


Figure 26: Overall analysis and batch transformation process with precedence TGGs

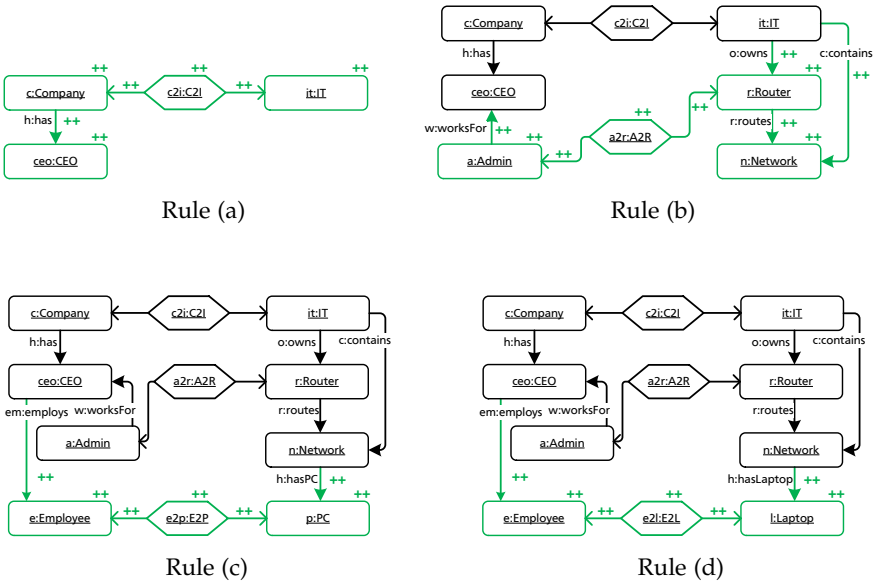


Figure 27: Local complete set of TGG rules for the batch transformation

6.1 PATHS AND PATTERNS

The concept of precedences encodes information about the transformation order between two or more nodes in a graph. Informally, a node x precedes another node y if y can only be transformed after x. In addition, two nodes x and y have the same precedence, if x and y can be transformed together by applying a single rule (i.e., within a single transformation step).

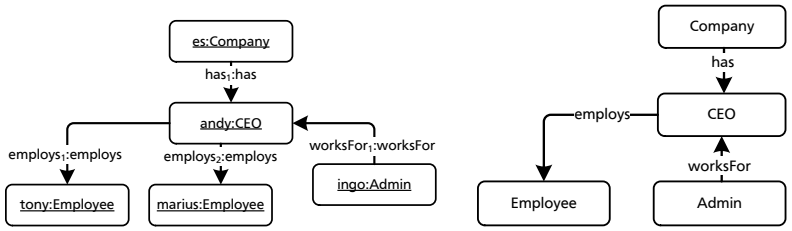
In order to calculate these precedences between nodes we have to analyze the rules in the TGG specification. As rules can be built of complex patterns, the analysis should not be restricted to incident nodes only (i.e., nodes that are directly connected), but retrieve information from all relevant pairs of nodes of each TGG rule. This allows the approach to detect all precedences in an input graph and not only those of directly connected elements.

Consequently, we introduce *paths*. A path defines a way through a graph starting with a specific node, traversing some other nodes and edges with or without their designated direction and ending at a specific node.

Definition 17 (Paths and Type Paths)
Let G *be a typed graph with type graph* TG.
A path p *between two nodes* $n_1, n_k \in V_G$ *is an alternating sequence of nodes and edges in* V_G *and* E_G*, respectively, denoted as* $p := n_1 \cdot e_1^{\alpha_1} \cdot n_2 \cdot \ldots \cdot n_{k-1} \cdot e_{k-1}^{\alpha_{k-1}} \cdot n_k$*, where* $\alpha_i \in \{+, -\}$ *specifies if an edge* e_i *is traversed from source* $s(e_i) = n_i$ *to target* $t(e_i) = n_{i+1}$ *(+), or in a reverse direction (–).*
A type path *is a path between node types and edge types in* V_{TG} *and* E_{TG}*, respectively.*
Given a path p*, its* type (path) *is defined as* $type_p(p) := type_V(n_1) \cdot type_E(e_1)^{\alpha_1} \cdot type_V(n_2) \cdot type_E(e_2)^{\alpha_2} \cdot \ldots \cdot type_V(n_{k-1}) \cdot type_E(e_{k-1})^{\alpha_{k-1}} \cdot type_V(n_k)$.

Example

Figure 28 depicts a typed graph and its type graph.



(a) Typed graph (b) Type graph

Figure 28: Typed and type graph of the source domain of our running example

Using the definition of paths, the following paths are determined by analyzing the typed graph of Fig. 28(a):

- $\texttt{es} \cdot \texttt{has}_1^+ \cdot \texttt{andy}$
- $\texttt{es} \cdot \texttt{has}_1^+ \cdot \texttt{andy} \cdot \texttt{employs}_1^+ \cdot \texttt{tony}$
- $\texttt{es} \cdot \texttt{has}_1^+ \cdot \texttt{andy} \cdot \texttt{employs}_2^+ \cdot \texttt{marius}$
- $\texttt{es} \cdot \texttt{has}_1^+ \cdot \texttt{andy} \cdot \texttt{worksFor}_1^- \cdot \texttt{ingo}$
- $\texttt{andy} \cdot \texttt{employs}_1^+ \cdot \texttt{tony}$
- $\texttt{andy} \cdot \texttt{employs}_2^+ \cdot \texttt{marius}$
- $\texttt{andy} \cdot \texttt{worksFor}_1^- \cdot \texttt{ingo}$
- $\texttt{andy} \cdot \texttt{has}_1^- \cdot \texttt{es}$
- $\texttt{tony} \cdot \texttt{employs}_1^- \cdot \texttt{andy}$
- $\texttt{tony} \cdot \texttt{employs}_1^- \cdot \texttt{andy} \cdot \texttt{has}_1^- \cdot \texttt{es}$
- $\texttt{tony} \cdot \texttt{employs}_1^- \cdot \texttt{andy} \cdot \texttt{employs}_2^+ \cdot \texttt{marius}$
- $\texttt{tony} \cdot \texttt{employs}_1^- \cdot \texttt{andy} \cdot \texttt{worksFor}_1^- \cdot \texttt{ingo}$
- $\texttt{marius} \cdot \texttt{employs}_2^- \cdot \texttt{andy}$
- $\texttt{marius} \cdot \texttt{employs}_2^- \cdot \texttt{andy} \cdot \texttt{has}_1^- \cdot \texttt{es}$
- $\texttt{marius} \cdot \texttt{employs}_2^- \cdot \texttt{andy} \cdot \texttt{employs}_1^+ \cdot \texttt{tony}$

- $\texttt{marius} \cdot \texttt{employs}_2^- \cdot \texttt{andy} \cdot \texttt{worksFor}_1^- \cdot \texttt{ingo}$
- $\texttt{ingo} \cdot \texttt{worksFor}_1^+ \cdot \texttt{andy}$
- $\texttt{ingo} \cdot \texttt{worksFor}_1^+ \cdot \texttt{andy} \cdot \texttt{has}_1^- \cdot \texttt{es}$
- $\texttt{ingo} \cdot \texttt{worksFor}_1^+ \cdot \texttt{andy} \cdot \texttt{employs}_1^+ \cdot \texttt{tony}$
- $\texttt{ingo} \cdot \texttt{worksFor}_1^+ \cdot \texttt{andy} \cdot \texttt{employs}_2^+ \cdot \texttt{marius}$

Each line denotes a path in the typed graph of Fig. 28(a). Considering for example the first line, we see that node `es` is connected to node `andy` via a direct edge $\texttt{has}_1$ that has to be traversed from source to target (+). Thus, symbol "·" is used as a separator.

In addition, the following paths are retrieved for the type graph of Fig. 28(b):

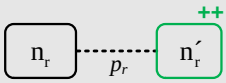
- $\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}$
- $\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$
- $\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $\texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $\texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$
- $\texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $\texttt{Employee} \cdot \texttt{employs}^- \cdot \texttt{CEO}$
- $\texttt{Employee} \cdot \texttt{employs}^- \cdot \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $\texttt{Employee} \cdot \texttt{employs}^- \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $\texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO}$
- $\texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $\texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$

Here, each line denotes a type path in the type graph of Fig. 28(b). Considering for example the first line, we see that type `Company` is connected to type `CEO` via a direct edge `has` that has to be traversed from source to target (+). Again, symbol "·" is used as a separator.

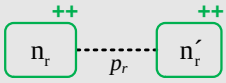
When analyzing rules for determining relevant precedence information, we need to find pairs of nodes that denote a *context* relationship or *create* relationship. Therefore, we define such patterns in TGG rules as follows:

Definition 18 (Relevant Node Creation Patterns)
Consider a $TGG = (TG, \mathcal{R})$ *with all its rules* $r \in \mathcal{R}$*, where* $r = (L, R) = (L_S \leftarrow L_C \rightarrow L_T, R_S \leftarrow R_C \rightarrow R_T)$.
The set $Paths_S$ *consists of all* acyclic paths *in* R_S *(note that* $L_S \subseteq R_S$*).*
The predicates $context_S : Paths_S \rightarrow \{true, false\}$ *and* $create_S : Paths_S \rightarrow \{true, false\}$ *in the source domain are defined as follows:*

- ***context***$_S(p_r) := \exists\, r \in \mathcal{R}$ *such that* p_r *is a path between two nodes* $n_r, n'_r \in R_S$: $(n_r \in L_S) \wedge (n'_r \in R_S \setminus L_S)$*, i.e., a rule* r *in* $\mathcal{R}$ *contains a path* p_r *which is isomorphic to the node creation pattern depicted in the diagram below.*



- ***create***$_S(p_r) := \exists\, r \in \mathcal{R}$ *such that* p_r *is a path between two nodes* $n_r, n'_r \in R_S$: $(n_r \in R_S \setminus L_S) \wedge (n'_r \in R_S \setminus L_S)$*, i.e., a rule* r *in* $\mathcal{R}$ *contains a path* p_r *which is isomorphic to the node creation pattern depicted in the diagram below.*



Example

Consider the source domain component of the TGG rules of our running example depicted in Fig. 27 In order to check if any representatives of the context or create pattern can be found in the rules, we build the set $Paths_S$ for each rule by collecting all paths according to Def. 17. Starting with Rule (a), the set $Paths_S$ consists of the following entries:

- $p_1 = c \cdot h^+ \cdot ceo$
- $p_2 = ceo \cdot h^- \cdot c$

Regarding path p_1, a path has been found in the source domain component of Rule (a) that denotes a connection between elements c and ceo. Thus, p_1 reflects a create pattern as both elements are in $R_S \setminus L_S$ (i.e., created elements). Analogously, all other paths are derived from the different rules.

Thus, the set $Paths_S$ consists of the following paths for Rule (b):

- $p_3 = c \cdot h^+ \cdot ceo$
- $p_4 = c \cdot h^+ \cdot ceo \cdot w^- \cdot a$
- $p_5 = ceo \cdot h^- \cdot c$

- $p_6 = \mathtt{ceo} \cdot \mathtt{w}^- \cdot \mathtt{a}$
- $p_7 = \mathtt{a} \cdot \mathtt{w}^+ \cdot \mathtt{ceo}$
- $p_8 = \mathtt{a} \cdot \mathtt{w}^+ \cdot \mathtt{ceo} \cdot \mathtt{h}^- \cdot \mathtt{c}$

Furthermore, the set Paths_S consists of the following paths for Rule (c):

- $p_9 = \mathtt{c} \cdot \mathtt{h}^+ \cdot \mathtt{ceo}$
- $p_{10} = \mathtt{c} \cdot \mathtt{h}^+ \cdot \mathtt{ceo} \cdot \mathtt{em}^+ \cdot \mathtt{e}$
- $p_{11} = \mathtt{c} \cdot \mathtt{h}^+ \cdot \mathtt{ceo} \cdot \mathtt{w}^- \cdot \mathtt{a}$
- $p_{12} = \mathtt{ceo} \cdot \mathtt{h}^- \cdot \mathtt{c}$
- $p_{13} = \mathtt{ceo} \cdot \mathtt{em}^+ \cdot \mathtt{e}$
- $p_{14} = \mathtt{ceo} \cdot \mathtt{w}^- \cdot \mathtt{a}$
- $p_{15} = \mathtt{e} \cdot \mathtt{em}^- \cdot \mathtt{ceo}$
- $p_{16} = \mathtt{e} \cdot \mathtt{em}^- \cdot \mathtt{ceo} \cdot \mathtt{h}^- \cdot \mathtt{c}$
- $p_{17} = \mathtt{e} \cdot \mathtt{em}^- \cdot \mathtt{ceo} \cdot \mathtt{w}^- \cdot \mathtt{a}$
- $p_{18} = \mathtt{a} \cdot \mathtt{w}^+ \cdot \mathtt{ceo}$
- $p_{19} = \mathtt{a} \cdot \mathtt{w}^+ \cdot \mathtt{ceo} \cdot \mathtt{h}^- \cdot \mathtt{c}$
- $p_{20} = \mathtt{a} \cdot \mathtt{w}^+ \cdot \mathtt{ceo} \cdot \mathtt{em}^+ \cdot \mathtt{e}$

Finally, the set Paths_S consists of the following paths for Rule (d):

- $p_{21} = \mathtt{c} \cdot \mathtt{h}^+ \cdot \mathtt{ceo}$
- $p_{22} = \mathtt{c} \cdot \mathtt{h}^+ \cdot \mathtt{ceo} \cdot \mathtt{em}^+ \cdot \mathtt{e}$
- $p_{23} = \mathtt{c} \cdot \mathtt{h}^+ \cdot \mathtt{ceo} \cdot \mathtt{w}^- \cdot \mathtt{a}$
- $p_{24} = \mathtt{ceo} \cdot \mathtt{h}^- \cdot \mathtt{c}$
- $p_{25} = \mathtt{ceo} \cdot \mathtt{em}^+ \cdot \mathtt{e}$
- $p_{26} = \mathtt{ceo} \cdot \mathtt{w}^- \cdot \mathtt{a}$
- $p_{27} = \mathtt{e} \cdot \mathtt{em}^- \cdot \mathtt{ceo}$
- $p_{28} = \mathtt{e} \cdot \mathtt{em}^- \cdot \mathtt{ceo} \cdot \mathtt{h}^- \cdot \mathtt{c}$
- $p_{29} = \mathtt{e} \cdot \mathtt{em}^- \cdot \mathtt{ceo} \cdot \mathtt{w}^- \cdot \mathtt{a}$
- $p_{20} = \mathtt{a} \cdot \mathtt{w}^+ \cdot \mathtt{ceo}$
- $p_{31} = \mathtt{a} \cdot w^+ \cdot \mathtt{ceo} \cdot \mathtt{h}^- \cdot \mathtt{c}$
- $p_{32} = \mathtt{a} \cdot \mathtt{w}^+ \cdot \mathtt{ceo} \cdot \mathtt{em}^+ \cdot \mathtt{e}$

Altogether, the predicate *context* holds for paths $p_4, p_6, p_{10}, p_{13}, p_{20}$, p_{22}, p_{25}, and p_{32} as these paths start at a context element and end at a created element. Thus, the predicate *create* is true for paths p_1 and p_2 as these paths start and end at created elements. All remaining paths do not represent a relevant pattern as they either connect two context elements (e.g., p_3) or start at a created element and end at a context element (e.g., p_7).

So far, rules can be analyzed and two different predicates have been introduced to check whether a path represents a context or a create pattern. In order to utilize this information to define a partial order for any given input graph, we have to lift these concepts from the instance level to the type level.

Therefore, two specific sets of interesting type paths, relevant for our analysis are defined:

Definition 19 (Type Path Sets)
The set TPaths_S *denotes all type paths of paths in* $Paths_S$ *(cf. Def. 18), i.e.,* $TPaths_S := \{tp \mid \exists\, p \in Paths_S \text{ such that } type_p(p) = tp\}$.
Thus, we define the restricted create type path set *for the source domain as*

- $TP_S^{create} := \{tp \in TPaths_S \mid \exists\, p \in Paths_S \wedge type_p(p) = tp \wedge create_S(p)\}$

and the restricted context type path set *for the source domain as*

- $TP_S^{context} := \{tp \in TPaths_S \mid \exists\, p \in Paths_S \wedge type_p(p) = tp \wedge context_S(p)\}$

Example

Based on the last example section, we compute the type for all 32 determined paths. Hence, the set TPaths_S consists of the following entries:

- $type_p(p_1) = tp_1 = \texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}$
- $type_p(p_2) = tp_2 = \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $type_p(p_3) = tp_3 = \texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}$
- $type_p(p_4) = tp_4 = \texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $type_p(p_5) = tp_5 = \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $type_p(p_6) = tp_6 = \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $type_p(p_7) = tp_7 = \texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO}$
- $type_p(p_8) = tp_8 = \texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $type_p(p_9) = tp_9 = \texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}$
- $type_p(p_{10}) = tp_{10} = \texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$
- $type_p(p_{11}) = tp_{11} = \texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $type_p(p_{12}) = tp_{12} = \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $type_p(p_{13}) = tp_{13} = \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$
- $type_p(p_{14}) = tp_{14} = \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $type_p(p_{15}) = tp_{15} = \texttt{Employee} \cdot \texttt{employs}^- \cdot \texttt{CEO}$
- $type_p(p_{16}) = tp_{16} = \texttt{Employee} \cdot \texttt{employs}^- \cdot \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $type_p(p_{17}) = tp_{17} = \texttt{Employee} \cdot \texttt{employs}^- \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $type_p(p_{18}) = tp_{18} = \texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO}$
- $type_p(p_{19}) = tp_{19} = \texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $type_p(p_{20}) = tp_{20} = \texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$
- $type_p(p_{21}) = tp_{21} = \texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}$
- $type_p(p_{22}) = tp_{22} = \texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$
- $type_p(p_{23}) = tp_{23} = \texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $type_p(p_{24}) = tp_{24} = \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $type_p(p_{25}) = tp_{25} = \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$
- $type_p(p_{26}) = tp_{26} = \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$

- $type_p(p_{27}) = tp_{27} = \texttt{Employee} \cdot \texttt{employs}^- \cdot \texttt{CEO}$
- $type_p(p_{28}) = tp_{28} = \texttt{Employee} \cdot \texttt{employs}^- \cdot \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $type_p(p_{29}) = tp_{29} = \texttt{Employee} \cdot \texttt{employs}^- \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}$
- $type_p(p_{30}) = tp_{30} = \texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO}$
- $type_p(p_{31}) = tp_{31} = \texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$
- $type_p(p_{32}) = tp_{32} = \texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$

Regarding Def. 19, the set TP_S^{create} consists of type paths tp_1, tp_2 and the set $TP_S^{context}$ consists of type paths $tp_4, tp_6, tp_{10}, tp_{13}, tp_{20}, p_{25}$ and tp_{32}. All remaining type paths can be neglected as they connect on the graph level either two context elements or start at a created element and end at a context element.

Together with these definitions, the so-called *precedence function* $\mathcal{PF}$ can be introduced that assigns certain type paths a specific *precedence symbol* to denote the concept of *precedence between nodes*, indicating that one node is a potential context node when transforming another node.

Definition 20 (Precedence Function $\mathcal{PF}_S$)
Let $\mathcal{P} := \{\lessdot, \doteq, \cdot\lightning\cdot\}$ *be the set of precedence relation symbols. Given a* $TGG = (TG, \mathcal{R})$ *and the restricted type path sets for the source domain* $TP_S^{create}, TP_S^{context}$*. The precedence function for the source domain* $\mathcal{PF}_S : \{TP_S^{create} \cup TP_S^{context}\} \to \mathcal{P}$ *is computed as follows:*

$$\mathcal{PF}_S(tp) := \begin{cases} \lessdot \textit{ iff } tp \in \{TP_S^{context} \setminus TP_S^{create}\} \\ \doteq \textit{ iff } tp \in \{TP_S^{create} \setminus TP_S^{context}\} \\ \cdot\lightning\cdot \textit{ otherwise} \end{cases}$$

Example

For our running example, $\mathcal{PF}_S$ consists of the following entries:
Rule (a): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}) = \doteq$ and
$\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}) = \doteq$
Rule (b): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}) = \lessdot$ and
$\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}) = \lessdot$
Rules (c) and (d): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^- \cdot \texttt{Employee}) = \lessdot$, $\mathcal{PF}_S(\texttt{Admin} \cdot \texttt{worksFor}^- \cdot \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee} = \lessdot)$ and $\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{employs}^- \cdot \texttt{Employee}) = \lessdot$

Restriction: As our precedence analysis depends on paths in rules of a given TGG, the presented approach requires TGG rules that are (weakly)[2] connected in each domain. This means that all pairs

2 Strongly connected domains would require that edges must not be traversed in reverse directions in order to find a path between two elements. This is not necessary as our definition of paths supports traversing edges in either direction (cf. Def. 17).

of nodes of each domain of a rule must be connected by a path. Hence, considering the source domain, the following must hold: $\forall r \in \mathcal{R}, \forall n, n' \in R_S : \exists p \in \text{Paths}_S$ between n and n'.

6.2 PRECEDENCE RELATIONS $\lessdot_S$ AND $\doteq_S$

Based on the precedence function $\mathcal{PF}_S$, relations $\lessdot_S$ and $\doteq_S^*$ can now be defined and used to topologically sort the nodes of a given input graph G_S and determine the sets of elements that can be transformed at each step in the algorithm.

As a first step we retrieve all paths in the input graph that contain relevant precedence information:

Definition 21 (Source Path Set)
For a given typed source graph G_S*, the* source path set *for the source domain is defined as follows:*
$P_S := \{p \mid p \textit{ is a path between } n, n' \in V_{G_S} \wedge \text{type}_p(p) \in \{TP_S^{create} \cup TP_S^{context}\}\}$.

Example

Considering the typed graph depicted in Fig. 28(a), the set P_S consists of the following entries:

- $\texttt{es} \cdot \texttt{has}_1^+ \cdot \texttt{andy}$
- $\texttt{es} \cdot \texttt{has}_1^+ \cdot \texttt{andy} \cdot \texttt{employs}_1^+ \cdot \texttt{tony}$
- $\texttt{es} \cdot \texttt{has}_1^+ \cdot \texttt{andy} \cdot \texttt{employs}_2^+ \cdot \texttt{marius}$
- $\texttt{es} \cdot \texttt{has}_1^+ \cdot \texttt{andy} \cdot \texttt{worksFor}_1^- \cdot \texttt{ingo}$
- $\texttt{andy} \cdot \texttt{employs}_1^+ \cdot \texttt{tony}$
- $\texttt{andy} \cdot \texttt{employs}_2^+ \cdot \texttt{marius}$
- $\texttt{andy} \cdot \texttt{worksFor}_1^- \cdot \texttt{ingo}$
- $\texttt{ingo} \cdot \texttt{worksFor}_1^+ \cdot \texttt{andy} \cdot \texttt{employs}_1^+ \cdot \texttt{tony}$
- $\texttt{ingo} \cdot \texttt{worksFor}_1^+ \cdot \texttt{andy} \cdot \texttt{employs}_2^+ \cdot \texttt{marius}$

Each entry denotes a path in the input graph that is typed either by a type path in TP_S^{create} or in $\text{TP}_S^{context}$, and, therefore, reflects either a create or a context pattern. These paths are all necessary paths to further determine an appropriate traversal order for processing the input graph G_S.

The next step is to derive the explicit precedences between nodes. So far, paths in P_S are those paths that either denote a *context* or *create* predicate (cf. Def. 18). In a first step, all pairs of nodes n, n' are added to relation $\lessdot_S$ if n could be (indirectly) used as context element when an appropriate rule is applied to transform n'.

Definition 22 (Precedence Relation $\lessdot_S$)
Given the precedence function $\mathcal{PF}_S$ for a given TGG*, and a typed source graph G_S. The precedence relation $\lessdot_S \subseteq V_{G_S} \times V_{G_S}$ for the source domain is defined as follows: $n \lessdot_S n'$ if there exists a path $p \in P_S$ between nodes n and n' such that $\mathcal{PF}_S(type_p(p)) = \lessdot$.*

Example

For the example graph (Fig. 28(a)), the following pairs constitute $\lessdot_S$: (`es` $\lessdot_S$ `ingo`), (`es` $\lessdot_S$ `tony`), (`es` $\lessdot_S$ `marius`), (`andy` $\lessdot_S$ `ingo`), (`andy` $\lessdot_S$ `tony`), (`andy` $\lessdot_S$ `marius`), (`ingo` $\lessdot_S$ `tony`), and (`ingo` $\lessdot_S$ `marius`).

Definition 23 (Relation $\dot{=}_S$)
Given the precedence function $\mathcal{PF}_S$ for a given TGG*, and a typed source graph G_S. The symmetric relation $\dot{=}_S \subseteq V_{G_S} \times V_{G_S}$ for the source domain is defined as follows: $n \dot{=}_S n'$ if there exists a path $p \in P_S$ between nodes n and n' such that $\mathcal{PF}_S(type_p(p)) = \dot{=}$.*

Relation $\dot{=}_S$ is symmetric, as for each path p between n and n' with the previously described properties, there always exists the reverse path p' starting at n' and ending at n. This is a direct consequence of the definition of creation pattern $create_S$ (cf. Def. 18).

Definition 24 (Equivalence Relation $\dot{=}^*_S$)
The equivalence relation $\dot{=}^*_S$ is the transitive and reflexive closure of the symmetric relation $\dot{=}_S$.

Example

For our example triple (Fig. 28(a)), the following equivalence classes constitute $\dot{=}^*_S$: {`andy`, `es`}, {`ingo`}, {`tony`}, and {`marius`}.

6.3 PRECEDENCE GRAPH

The previous section described the static compile-time precedence analysis completely (cf. step II of Fig. 26). We are now going further (cf. step III of Fig. 26) to determine the so-called *precedence graph*

which can be computed for a given input graph. Such a precedence graph sorts the equivalence classes defined by relation $\doteq_S^*$ along their precedences defined by relation $\lessdot_S$.

Definition 25 (Precedence Graph $\mathcal{PG}_S$)
Given a TGG*, a source graph* G_S *and its corresponding equivalence relation* $\doteq_S^*$ *and precedence relation* $\lessdot_S$*. The precedence graph* $\mathcal{PG}_S$ *is a graph constructed as follows:*
(i) The equivalence relation $\doteq_S^*$ *is used to partition* V_{G_S} *into equivalence classes* $EQ_1, \dots EQ_n$ *which serve as the nodes of* $\mathcal{PG}_S$*, i.e.,* $V_{\mathcal{PG}_S} := \{EQ_1, \dots, EQ_n\}$.
(ii) The edges in $\mathcal{PG}_S$ *are defined as follows:* $E_{\mathcal{PG}_S} := \{e \mid s(e) = EQ_i,\ t(e) = EQ_j : \exists\, n_i \in EQ_i, n_j \in EQ_j$ *with* $n_i \lessdot_S n_j\}$.

Remark: $\mathcal{PG}_S$ defines a partial order over equivalence classes whenever the precedence relation $\lessdot_S$ is acyclic (this also requires $\lessdot_S$ to be irreflexive). This is a direct consequence of Def. 25. To statically check whether cyclic precedence graphs are possible for a given TGG specification, a static analysis will be introduced as an addition in Chap. 14.

Example

Figure 29 repeats the input graph G_S that will be transformed.

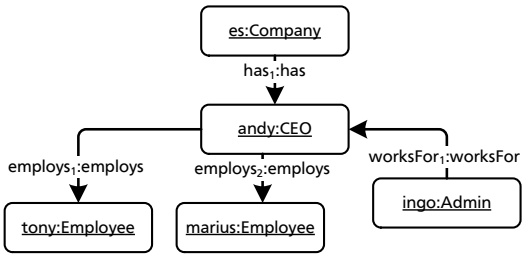


Figure 29: Input graph G_S

Together with the precedence function $\mathcal{PF}_S$ derived by analyzing all Rules (a) to (d) (cf. Fig. 27), the relation $\lessdot_S$ is constituted as follows:
(`es` $\lessdot_S$ `ingo`), (`es` $\lessdot_S$ `tony`), (`es` $\lessdot_S$ `marius`), (`andy` $\lessdot_S$ `ingo`), (`andy` $\lessdot_S$ `tony`), and (`andy` $\lessdot_S$ `marius`).

Furthermore, the following equivalence classes constitute $\doteq_S^*$:
{`andy`, `es`}, {`ingo`}, {`tony`}, and {`marius`}

The corresponding precedence graph $\mathcal{PG}_S$ is constructed from our input graph G_S as follows:

- $V_{\mathcal{PG}_S} = \{EQ_1 = (\texttt{andy}, \texttt{es}),\ EQ_2 = (\texttt{ingo}),\ EQ_3 = (\texttt{tony}),\ EQ_4 = (\texttt{marius})\}$

- $E_{\mathcal{PG}_S} = \{e_1, e_2, e_3, e_4, e_5\}$ where $s(e_1) = EQ_1$, $t(e_1) = EQ_2$ as `es` $\lessdot_S$ `ingo` and `andy` $\lessdot_S$ `ingo`, $s(e_2) = EQ_1$, $t(e_2) = EQ_3$ as `es` $\lessdot_S$ `tony` and `andy` $\lessdot_S$ `tony`, and $s(e_3) = EQ_1$, $t(e_3) = EQ_4$ as `es` $\lessdot_S$ `marius` and `andy` $\lessdot_S$ `marius`, $s(e_4) = EQ_2$, $t(e_4) = EQ_3$ as `ingo` $\lessdot_S$ `tony`, and finally $s(e_5) = EQ_2$, $t(e_5) = EQ_4$ as `ingo` $\lessdot_S$ `marius`

The precedence graph is depicted in Fig. 30 with the equivalence classes surrounded by dashed lines.

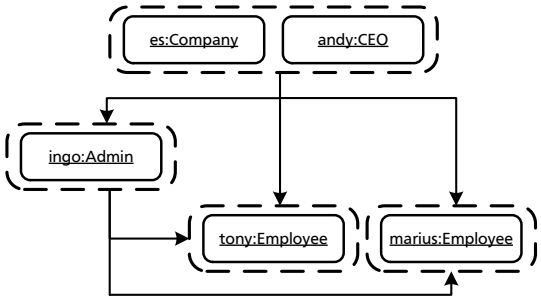


Figure 30: Precedence graph $\mathcal{PG}_S$ for the input graph G_S

7

BATCH CONTROL ALGORITHM

In this chapter, the precedence driven batch algorithm is presented (cf. Algorithm 1). This algorithm utilizes the introduced precedence function $\mathcal{PF}_S$ and computes a precedence graph $\mathcal{PG}_S$ to define a partial order for the input graph. This allows for the efficient batch transformation of models. Regarding the overall process depicted in Fig. 26 we are now discussing steps IV and V.

7.1 THE CONTROL ALGORITHM

For a forward transformation (a backward transformation works analogously), the input for the algorithm is a source graph G_S and the statically derived precedence function for the source domain $\mathcal{PF}_S$.

Algorithm 1 Precedence TGG Batch Algorithm

```
1:  procedure TRANSFORM(G_S, PF_S)
2:    PG_S ← BUILDPRECEDENCEGRAPH(G_S, PF_S)
3:    while (PG_S contains equivalence classes) do
4:      readyNodes ← all nodes in equiv. classes in PG_S
                     without incoming edges
5:      select node n from readyNodes
6:      transformedNodes ← CHOOSEANDAPPLYRULE(n) ▷ utilize rule selection
                                                   and application capabilities of Algo. O
7:      if transformedNodes ≠ ∅ then
8:        PG_S ← remove all nodes in transformedNodes from PG_S
9:      else if transformedNodes = ∅ then
10:       terminate with exception
11:     end if
12:   end while
13:   return G_S ← G_C → G_T
14: end procedure
```

Procedure TRANSFORM determines a graph triple $G_S \leftarrow G_C \rightarrow G_T$ as output. The first step (line (2)) of the algorithm produces the appropriate precedence graph $\mathcal{PG}_S$ for the input graph G_S according to Def. 25. Note that the procedure BUILDPRECEDENCEGRAPH will terminate with an exception if there is a cycle in the precedence graph and it is thus impossible to sort the elements of the source graph according to their dependencies. Starting on line (3), a while-loop iterates over equivalence classes in $\mathcal{PG}_S$ until there are none left. In the while-loop, the set *readyNodes* contains all nodes that can be transformed next, i.e., whose context elements have already been trans-

formed (line (4)). This set is determined by taking all nodes in the equivalence classes of $\mathcal{PG}_S$, which do not have incoming edges (dependencies). Next, one of the nodes n in *readyNodes* is selected on line (5). On line (6), the procedure CHOOSEANDAPPLYRULE is used to determine and filter the rules, allowing for user input or choosing arbitrarily from the applicable rules. If a rule was successfully chosen and applied to transform n on line (6), a non-empty set of *transformedNodes* is returned that is used to update $\mathcal{PG}_S$ on line (8). In this case, the while-loop is repeated with the updated and thus "smaller" $\mathcal{PG}_S$. If *transformedNodes* is empty, we know that node n has not been transformed and that the algorithm has hit a dead-end. This can only happen for TGGs that violate the *Local Completeness Criterion* (cf. Sect. 4.6.1) and are *not* in the class of supported TGGs or in cases where the input graph G_S is not schema-compliant regarding the source domain of the TGG specification.

The method call on line (6) denotes the reuse of the previously reviewed Algorithm O (cf. Sect. 4.6). This method is similar to the method TRANSFORM(N) of the context-driven algorithm without the recursive call on line (19) as all context elements are always already transformed (cf. proof in Sect. 7.3).

Example

To demonstrate the presented algorithm, we apply a forward transformation for the source graph G_S depicted in Fig. 31.

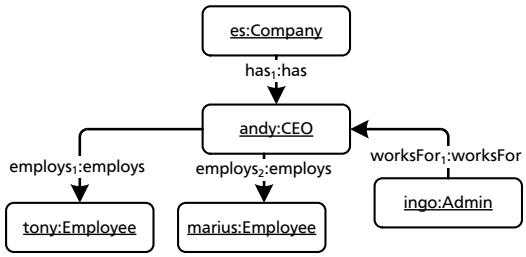


Figure 31: Input graph G_S

Additionally, the precompiled precedence function $\mathcal{PF}_S$ consists of the following entries:

- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}) = \doteq$
- $\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}) = \doteq$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}) = \lessdot$
- $\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^- \cdot \texttt{Employee}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Admin} \cdot \texttt{worksFor}^- \cdot \texttt{CEO} \cdot \texttt{employs}^+ \cdot \texttt{Employee}) = \lessdot$
- $\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{employs}^- \cdot \texttt{Employee}) = \lessdot$

On line (2), the precedence graph $\mathcal{PG}_S$ for G_S is built (depicted in Fig. 32). $\mathcal{PG}_S$ is acyclic, hence the transformation can continue.

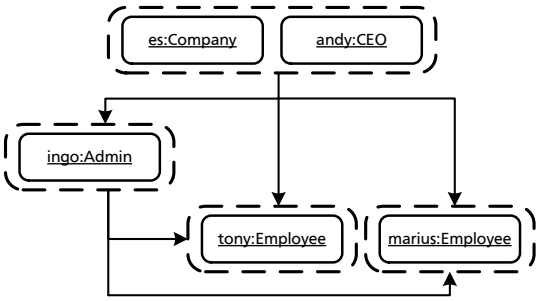


Figure 32: Precedence graph $\mathcal{PG}_S$ for the input graph G_S

On line (4), the set *readyNodes* is determined, consisting in this case of the nodes `es` and `andy` from the same equivalence class of $\mathcal{PG}_S$. On line (5) `es` or `andy` is chosen randomly, and in either case, the only candidate rule is Rule (a) (Fig. 21), which can be directly applied on line (6). Again in either case, *transformedNodes* contains both nodes as Rule (a) transforms `es` and `andy` simultaneously. $\mathcal{PG}_S$ is updated on line (8) to consist of three equivalence classes `ingo`, `tony`, and `marius`. In the second iteration through the while-loop, *readyNodes* now contains the node `ingo` and selects this node in line (5). Applying Rule (b) on line (6) puts `ingo` in *transformedNodes*, $\mathcal{PG}_S$ is updated on line (8) to now contain only `tony` and `marius`. In the third iteration, *readyNodes* contains `tony` and `marius`. On line (5) `tony` could be randomly selected first and Rule (d) is chosen (arbitrarily or via user input) to be applied on line (6). After updating $\mathcal{PG}_S$ again, only `marius` remains untransformed. Similarly to the penultimate iteration, Rule (c) is selected and applied this time. Updating $\mathcal{PG}_S$ on line (8) empties the precedence graph, which terminates the while-loop on line (3). The created graph triple depicted in Fig. 33 is returned on line (13).

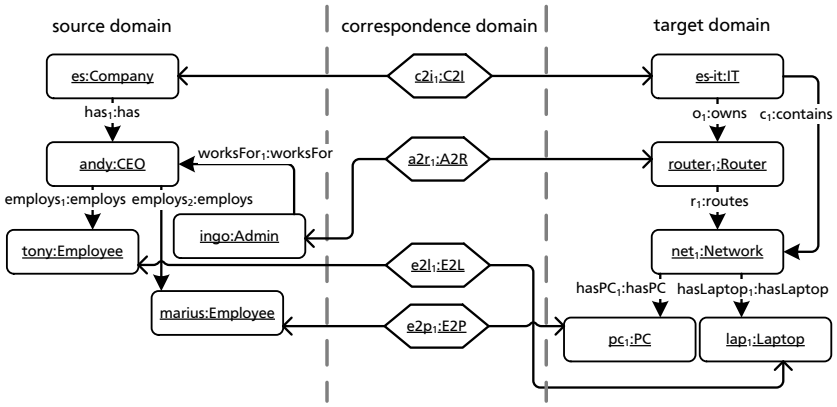


Figure 33: Resulting graph triple $G = G_S \leftarrow G_C \rightarrow G_T$ after forward transforming G_S

7.2 FORMAL PROPERTIES

In the following we will prove that the presented algorithm retains all formal properties stipulated in [SK08] (cf. Sect. 4.3) and proved for the context-driven algorithm of Sect. 4.6 in [KLKS10].

Theorem 1
Algorithm 1 is correct, complete and efficient for any local complete TGG.

Proof
Correctness:
If the algorithm returns a graph triple, i.e., does not terminate with an exception, it was able to find a sequence of source rules $r_{1_S}, r_{2_S}, \ldots, r_{n_S}$ that would build the given source graph G_S and, thus, the corresponding sequence of forward rules $r_{1_F}, r_{2_F}, \ldots, r_{n_F}$ that transform the given source graph (Def. 13). The *Decomposition and Composition Theorem* of [EEE+07] guarantees that it is possible to compose the sequence $r_{1_S}, r_{2_S}, \ldots, r_{n_S}, r_{1_F}, r_{2_F}, \ldots, r_{n_F}$ to the sequence of TGG rules $r_1, r_2, \ldots, r_n$ which proves that the resulting graph triple is consistent, i.e., $G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$. □

Completeness:
Algorithm 1 transforms nodes with the same techniques (e.g., dangling edge check) as Algorithm O [KLKS10] (cf. Sect. 4.6) as it reuses its rule selection and application capabilities. Therefore, no novel properties have to be proved regarding the actual selection and application of appropriate rules in order to process a certain node `n`.
The algorithm of [KLKS10] determines the processing order in an on-demand manner. In contrast, the here presented algorithm processes nodes in a *semi-fixed order* that is determined in advance. The sequence is only *semi-fixed* as the transformation algorithm can always choose a node arbitrarily for processing from the set `readyNodes`. The algorithm uses the precedence graph $\mathcal{PG}_S$ to determine this processing order. According to the definition of precedence graphs (cf. Def. 25), it is guaranteed that the context of every node in the set `readyNodes` is always already transformed. Assume that the context-driven Algorithm O would be forced to process all nodes in exactly this determined order. As a direct consequence, the context-driven Algorithm O would never run into situations where unprocessed context nodes have to be processed on-demand. Hence, the on-demand handling of unprocessed context nodes can be safely removed without compromising the completeness property. This directly leads to the new Algorithm 1. Consequently, all arguments from [KLKS10] referring to completeness can be transferred to Algorithm 1: this algorithm either returns a consistent graph triple whenever a such a

graph triple exists or terminates the transformation with an exception in cases where the TGG does not belong to the class of supported TGGs (e.g., not local complete). □

Efficiency:
At first, the precedence graph $\mathcal{PG}_S$ is computed on line (2), which requires to find all instances of type paths defined in the precedence function $\mathcal{PF}_S$ of the given TGG. As this requires a pattern matching process, we have to discuss how costly it is to find all these instances. We regard k as the maximum number of elements of all rules. In the worst case, all rules are specified over a complete graph (i.e., fully-connected). Paths can be at most of length k as we consider acyclic type paths only. Determining all paths of any length from 1 to k for a node requires at most n^k elements, where n is the number of nodes in our instance model. This denotes the typical worst-case complexity of pattern matching [SK08]. In our case, the exponent is decreased by one (i.e., n^{k-1}), because one of the elements of the path is already bound as it is the element that is just regarded. As this has to be repeated for all nodes (i.e., n times), we estimate that building $\mathcal{PG}_S$ costs at most $n \cdot n^{k-1} = n^k$.
The while-loop starting on line (3) iterates through $\mathcal{PG}_S$, and removes in every iteration at least one transformed node from $\mathcal{PG}_S$. Thus, the while-loop has in the worst-case (i.e., all equivalence classes in $\mathcal{PG}_S$ consist of exactly one node) n cycles. Within the while-loop, we select equivalence classes without incoming edges on line (4). Assuming that an efficient data structure, such as a heap, is used to represent $\mathcal{PG}_S$, this selection can be achieved in $\mathcal{O}(\log(n))$ by iterating through $\mathcal{PG}_S$. As argued in [KLKS10], transforming a node, i.e., checking all conditions and performing pattern matching (line (6)), is assumed to run in $\mathcal{O}(n^{k-1})$ (cf. Def. 16). Updating $\mathcal{PG}_S$ (e.g., efficiently implemented as a heap) on line (8) requires traversing all successor nodes which is at most $\log(n)$. Summarizing, we obtain: $n^{k-1} + n \cdot (\log(n) + n^{k-1} + \log(n))) \in \mathcal{O}(n^k)$. □

Remark: In practice, the batch transformation algorithm would also have to build the precedence graph of the target domain $\mathcal{PG}_T$ in order to allow for further model modifications in both domains. This, of course, induces additional runtime costs that have a similar complexity compared to those building the source domain precedence graph. Nevertheless, this combined complexity will be in sum still in the same complexity class.

As TGGs are symmetric [HEO+11], all arguments can be transferred analogously to backward transformations.

8

EVALUATION OF THE BATCH ALGORITHM

In this part one of the two major results of this thesis were proposed: Based on the formal theory of TGGs, we presented a so-called precedence analysis in Chap. 6. This analysis extracts at compile-time information from TGG rules that is suitable for defining a partial order on input graphs. This partial order will then be used by the novel control algorithm (cf. Chap. 7) to determine an appropriate traversal order for the input graph.

Algorithm 1 is an equivalent compared to the context-driven batch algorithm of [KLKS10] (cf. Sect. 4.6). This means that the algorithm is able to handle input graphs in an efficient and distinct transformation sequence. Altogether, compared to other approaches and especially the Klar algorithm (cf. Sect. 4.6), Algorithm 1 has the following benefits:

- **No bookkeeping efforts**: The algorithm does not require any bookkeeping of already transformed nodes. Whenever the precedence graph determines a node (more specific equivalence class) has no incoming dependencies, it is guaranteed that this node is actually ready for transformation.
- **No recursion stack**: Context-driven algorithms rely on recursion and the on-demand transformation of context elements. The presented Algorithm 1 works in a top-down manner and, therefore, does not employ a costly recursion stack (w.r.t. memory consumption).
- **Single DEC check only**: Compared to the Klar algorithm, only a single DEC check has to be used throughout the transformation process. This check is employed in the chooseAndApply method and guarantees that all remaining untransformed edges are still transformable after applying a selected rule. In the case of [KLKS10], an additional DEC check has to be employed after all context elements have been transformed. This was necessary, as transforming context elements may have changed certain conditions.
- **Early cycle detection**: After having built the precedence graph it can easily be checked whether $\mathcal{PG}_S$ is acyclic or not. In the case that $\mathcal{PG}_S$ is cyclic, we know that a dependency cycle exists and, hence, the algorithm will not be able to process the given input graph G_S. For the context-driven algorithm, such a cycle check is employed on demand whenever a node is processed. Hence, it is

possible that a cycle will be detected at the end of a (potentially costly) transformation process and result in an exception.

- **Improved operational rules**: Finally, as the construction of precedence graphs guarantees that context elements are always already transformed regarding any node in `readyNodes`, we can directly match context elements and apply a rule in one single operationalized rule (per TGG rule). Considering the context-driven approach of [KLKS10], the context elements have to be determined first and transformed recursively afterwards. Therefore, it was necessary to retrieve the context elements in a separate method first. From a technical point of view it is now possible to generate lean operational rules.

Besides these benefits, the presented algorithm still leaves some space for improvements: The precedence relations presented in this thesis treat nodes as first class objects only. Therefore, ordering information can only be retrieved between pairs of nodes and not between pairs of edges or mixed pairs. This can be solved by extending the precedence analysis in a straight-forward manner with edge-considering capabilities.

Thus, the presented precedence analysis and, therefore, the control algorithm as well, suffer from efficiency problems with collecting all paths. Since rules may encode the same precedence between two nodes with different paths, all these paths will be searched when building the precedence graph. To overcome this, we will present an appropriate improvement in Chap. 12.

Regarding the expressiveness of our approach, $\lessdot/\doteq$ - conflicts (cf. Def. 20) are prohibited. This induces a restriction of the allowed TGG rules. As future work, one possible strategy could be to introduce a new partial relation that combines both. The result would be that potential precedences (e.g., element n must either be processed before n' ($n \lessdot n'$) or together ($n \doteq n'$)) are guaranteed to be handled at first. From a formal point of view, we still have to show that in cases the algorithm processes n, n' at once, no problems are caused, which means that other precedences of n' must have been processed in advance.

Finally, a major language feature of [KLKS10] has been neglected so far: negative application conditions (NACs). Those conditions allow for defining global constraints that have to be fulfilled by consistently integrated graph triples and, hence, influence the transformation process as they help to prevent the algorithm from building schema-incompliant models. First thoughts regarding NACs in precedence TGGs show that NACs can be handled similarly as in the context-driven case of [KLKS10]. This seems to be true, as NACs in the sense of [KLKS10] influence only the creation of output models. As our approach utilizes only the input model for controlling the transfor-

mation sequence, NACs do not influence this sorting and, hence, do not introduce new conditions regarding our precedences.

Part III

INCREMENTAL MODEL SYNCHRONIZATION WITH TGGS

In this part, the main contribution of this thesis will be introduced. Therefore, based on our contribution of [LAVS12b] we propose in the following chapters:

(i) an extension to the theory of deriving operational rules in: so-called *inverse rules* will be used by the incremental algorithm to revoke the effects of a previous rule application (cf. Chap. 9).

(ii) an incremental control algorithm in Chap. 10 that uses the previously presented precedence analysis to determine the influences of model changes appropriately.

Regarding the challenges of bidirectional model synchronization (cf. Chap. 1), this incremental control is be able to (i) efficiently perform the synchronization while (ii) retaining information as much as possible. As this algorithm also complies to all formal properties, all challenges are mastered.

Figure 34 depicts how the algorithm to be presented in this part founds on Algorithm 1 of Part ii and the reused part of [KLKS10]. Again, the shaded boxes denote those components that are treated as black-boxes regarding the presentation of the incremental algorithm.

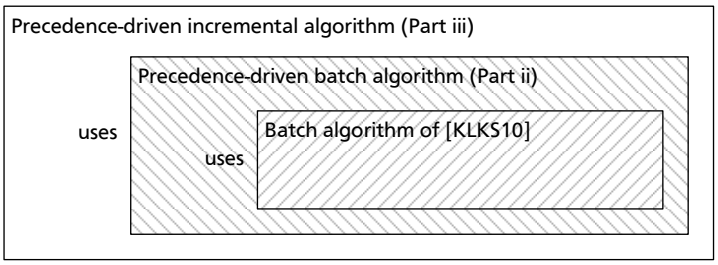


Figure 34: Schematic overview of building the incremental algorithm

9 INCREMENTAL MODEL CHANGES

In this chapter, we discuss (i) typical incremental model changes applied to our running example and (ii) the extension of the derivation of operational rules in order to revoke the effects of previous rule applications.

Another important aspect of incremental model changes is to understand that it is a hard task to efficiently compute all elements that are (potentially) affected by such a change and, therefore, have to be reconsidered for propagating the change. As mentioned in the introduction already, different scenarios are possible to process (potentially) affected elements: (i) revoke a previous transformation of a potentially affected element, and retransform it; (ii) check if all conditions of the previously applied transformation are still satisfied (a.k.a. consistency check). If this is the case, nothing has to be done, or otherwise the previous transformation will be revoked and the element has to be retransformed later. Furthermore, both options can be improved by reusing as much elements as possible regarding the to-be-deleted elements of the opposite domain and/or by determining the set of potentially affected elements as close as possible compared to the set of actually affected elements. For reasons of practicality, we restrict ourselves to option (i) without optimization. However, future work should cover these aspects to further improve this approach regarding efficiency and information preservation capabilities.

9.1 INTRODUCTION OF MODEL CHANGES

So far, we discussed only batch transformations for given input models. Hence, we have to extend our running example with concrete scenarios of incremental model changes. These scenarios will then be used in Chap. 10 to discuss the mode of operation of our incremental control algorithm.

Example

To extend our running example with a concrete model change scenarios, we introduce an additional element: a `Division`. A `Company` is split into different `Divisions` and each of these `Division` has its own set of `Employees` and `Admins`. An exemplary `Company` and integrated `IT` structure are depicted in Fig. 35

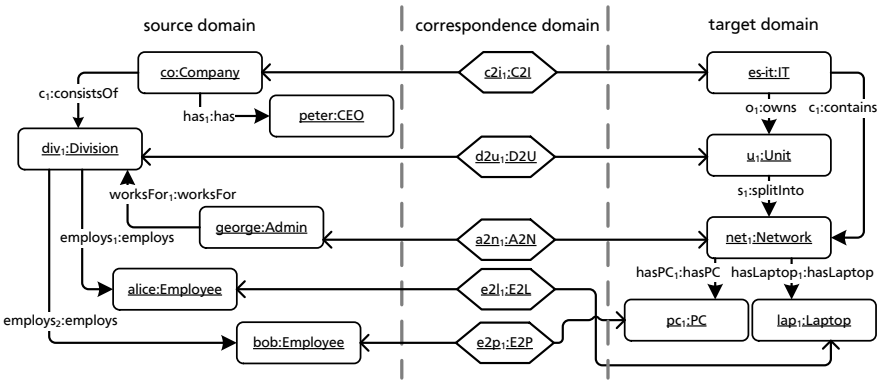
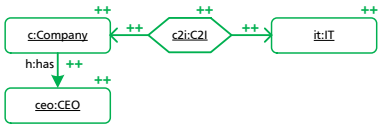
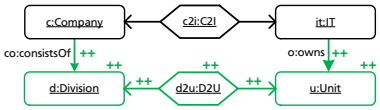


Figure 35: Extended integration of a `Company` and `IT`

The idea is that a `CEO` controls all `Divisions`, but these `Divisions` are an intermediate level for aggregating groups of `Employees` and `Admins`. Therefore, the set of rules has to be slightly adjusted in order to comply to the new structural requirements. Figure 36 depicts the new set of TGG rules.

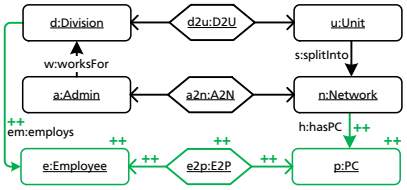


Rule (a)

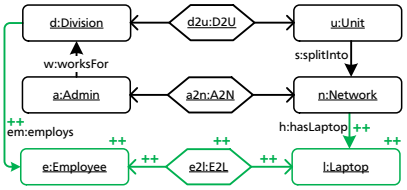


Rule (b)

Rule (c)



Rule (d)



Rule (e)

Figure 36: Adjusted set of rules for incremental model changes

- Rule (a) depicted in Fig. 36(a) integrates a `Company` together with a `CEO` in the source domain and an `IT` infrastructure in the target domain.
- The additional rule introducing a division to this running example is depicted in Fig. 36(b). Here, an existing `Company` is ex-

panded with a `Division` in the source domain, while a `Unit` object is created in the target domain.

- In Fig. 36(c), Rule (c) equips a division with an `Admin` who is in charge of maintaining a `Network`.
- Rule (d) depicted in Fig. 36(d) assigns an `Employee` to a `Division` and provides him or her with a `PC` that will be attached to an existing `Network`.
- Finally, Rule (e) depicted in Fig. 36(e) assigns an `Employee` to a `Division` and provides him or her with a `Laptop` that will again be connected to an existing `Network`.

The last two rules (i.e., Rules (d) and (e)) are important to highlight why information preservation during synchronization is especially important for non-functional TGGs as they allow for processing an `Employee` with two different rules and, hence, influence the resulting graph triple.

Together with the concepts of a precedence analysis presented in Chap. 6, we retrieve the following precedence function $\mathcal{PF}_S$ for the source domain:
Rule (a): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}) = \doteq$ and
$\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}) = \doteq$
Rule (b): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{consistsOf}^+ \cdot \texttt{Division}) = \lessdot$
Rule (c): $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}) = \lessdot$
Rule (d): $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Employee}) = \lessdot$ and
$\mathcal{PF}_S(\texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Employee}) = \lessdot$
Rule (e): $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Employee}) = \lessdot$ and
$\mathcal{PF}_S(\texttt{Admin} \cdot \texttt{worksFor}^+ \cdot \texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Employee}) = \lessdot$

Changes and Change Recognition

To demonstrate how *incremental changes* have to be propagated by an appropriate control algorithm, different changes have to be handled. According to OMG's MOF Facility Object Lifecycle (MOFFOL) [OMG10] three different types of change operations have to be distinguished: (1) *add new elements*, (2) *delete existing elements*, or (3) *move existing elements*. Typically, one would expect to have a fourth operation, namely *modify an existing element*. This thesis is restricted to additions and deletions in models. Coping with modifications (such as moving an element or modifying its content) is left to future work for two reasons. At first, as only typed graphs without attributes are

supported,[1] modifying an element (except for changing edges) seems to be no actual use case. Second, in order to propagate modifications they have to be recognized first.

Changes can be recognized in two different ways. First, models are modified "online" which means that any adjustment is traced and can be replayed afterwards. This can be achieved for example with the approach of "Coupled Evolution of Metamodels and Models" (COPE) [HBJ09, Her11], where a distinct Eclipse plugin listens for model and metamodel changes and traces them. Unfortunately, this imposes the requirement to modify models in a fixed environment only (e.g., a specific development environment).

Due to the fact that it is our designated goal not to enforce future users to use another tool, model changes have to be discovered in another way which utilizes a model diff. A model diff is a software that compares two or more versions of a model and derives the applied changes a-posteriori. Unfortunately, standard diffs such as EMF Compare [BP08] or UML Diff [XS05] have problems recognizing changes in elements because internal heuristics have to be trained or adapted to the actual model type under consideration. Of course, if the model under consideration is supplied with unique identifiers for each element, this comparison can be achieved more reliably. But again, the metamodels of the domains which are to be integrated are not allowed to be adjusted.

One of the most flexible but commercial approach is the so-called SiDiff [SG08]. Here users can define specific characteristics for each object type and how to determine if an element has changed.

Due to this complexity,[2] it was decided to support *additions* and *deletions* of elements only (up to now). Every element change, therefore, will be recognized as a sequence of deleting the old and adding the updated element. Although this does not consider the sophisticated reuse of elements (as for example supposed by Greeneyer in [GPR11, GR11]) we are able to guarantee that the synchronized models are in a consistent state and that the algorithm is able to handle all changes.

As described previously, the presented synchronization approach will be able handle additions and deletions of model elements. The challenge is to perform the update in an *efficient* manner and to avoid information loss by retaining unaffected elements of the models. De-

1 Please note that attributes are irrelevant regarding our precedence analysis. As long as an element will be identified as changed (regardless in which sense), our approach will be able to determine the (potentially) affected elements. Hence, it is not a restriction of our approach that attributes are not considered, but a simplification for reasons of improved legibility.

2 For a more detailed introduction into the research area of model differencing, the reader is referred for example to the contributions of Kolovos, Förtsch, or Westfechtel: [KPP06, FW07, Wes10].

termining such an update sequence is a difficult task because transformations of deleted elements and their dependencies, as well as transformations of potential dependencies of newly added elements must be revoked [HEO+11]. The challenge is to identify such dependent elements in the model and to undo their previous transformation taking all changes into account.

To demonstrate that the here presented control algorithm is able to exactly achieve this, exemplary scenarios in the domain of our running example are defined. Each of these scenarios (i) reflects typical model changes according to our experience and (ii) will be used as an input for our incremental control algorithm in Chap. 10. In the following, we introduce four different incremental scenarios, each based on the model depicted in Fig. 35.

Example

Scenario 1: A `Company` consists of a single `Division` with a set of `Employees` and one `Admin`. The most trivial case of applying a change to the model is to make an `Employee` redundant and, therefore, remove this element from the model. Figure 37 depicts this scenario.

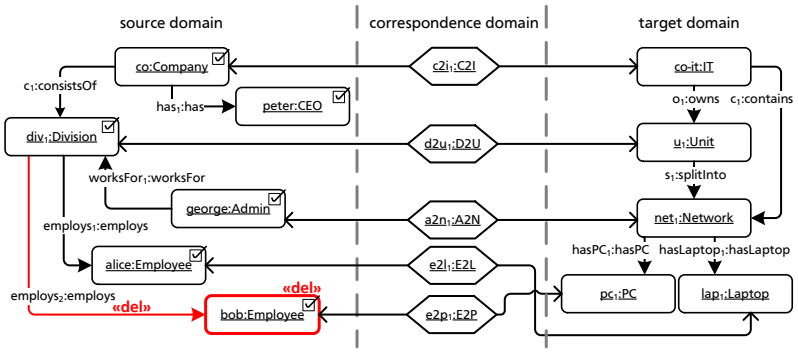


Figure 37: Scenario 1: Dismiss an `Employee`

Note that all transformed elements are equipped with a checkbox (☑). Of course, all elements of the triple are transformed as the triple depicts an existing model state, but checkboxes are only visible for source domain elements as a forward transformation (synchronization) is considered only. Thus, we depict checkboxes for nodes only to improve readability. This is sufficient, as edges are transformed in our approach always together with nodes. Again, all arguments are symmetric and, hence, can be applied to the target domain and backward transformations (synchronization) as well. The stereotype «`del`» denotes that this element has been deleted. This distinction is necessary as models are depicted intertwined together with the appropriate model changes. Furthermore, the stereotype «`add`» denotes elements that are newly added to the model.

Scenario 2: Another basic scenario is to hire a new `Employee` and equip him or her with a `PC` or `Laptop`. This scenario is depicted in Fig. 38, where an additional `Employee` has been added to the source domain.

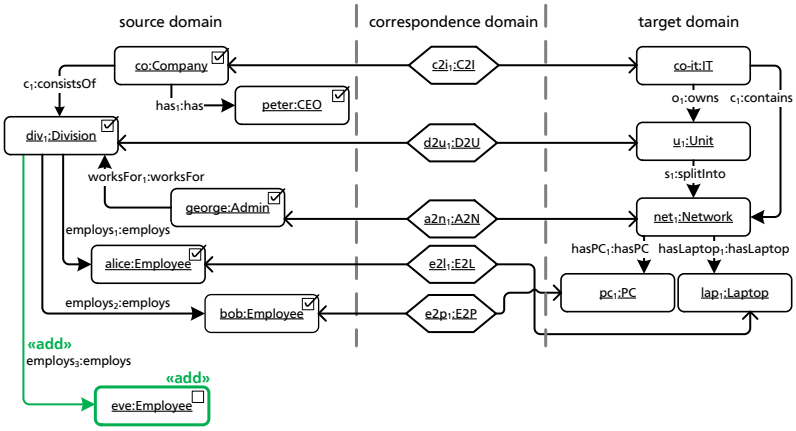


Figure 38: Scenario 2: Hire an additional `Employee`

Scenario 3: A third scenario (depicted in Fig. 39) would be to exchange the `CEO` of a `Company`. In such a case the existing `CEO` is removed, while a new `CEO` is introduced to the `Company` structure. As a `Company` is founded together with a `CEO`, a complete resynchronization has to be triggered. Such a resynchronization has to completely rebuild the target domain model as all elements in the source domain depend on the `CEO`. Nevertheless, in practical scenarios it is to be expected that changes occur more frequently on leaves, and rarely on root elements. Thus, a complete reconstruction will be rarely necessary.

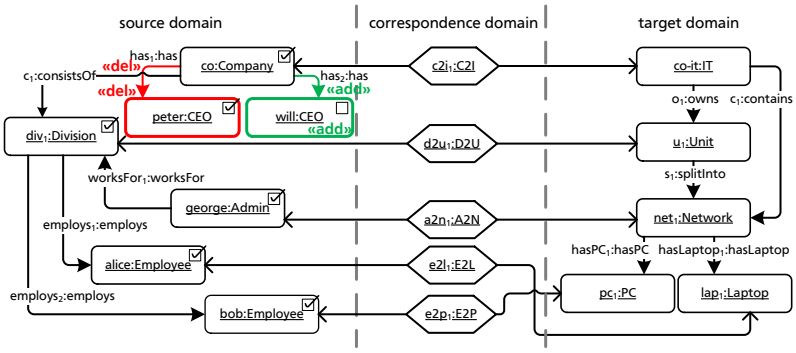


Figure 39: Scenario 3: Exchange the `CEO`

Scenarion 4: A fourth scenario is to employ an additional `Admin` in one of the `Divisions` (depicted in Fig. 40). As the maintenance capacity will be increased, additional `Networks` will be available. Hence established integrations between `Employees` and their `PCs` and `Laptops` have to be revised. Reconsidering dependent elements is necessary

as new options arise to process the `Employees`. More specifically, as more `Networks` will exist after the additional `Admin` has been integrated, the `Laptops` and `PCs` can be assigned differently now. At a first glance, this effort is unnecessary as the graph triple would still be consistently integrated even if the additional `admin` has been processed only as this transformation would not invalidate the previous transformations of the `Employees`. Unfortunately, this is not always the case. Introducing new elements to a model may also result in additional edges, which can only be processed if a still consistently integrated element is retransformed afterwards (cf. dangling edge condition in Sect. 4.6). The same is true if edges are removed. Again, it is possible that the previously satisfied dangling edge condition now fails on certain elements. Therefore, dependent elements have to be retransformed as only this action guarantees that previously made, now wrong choices, are corrected appropriately.

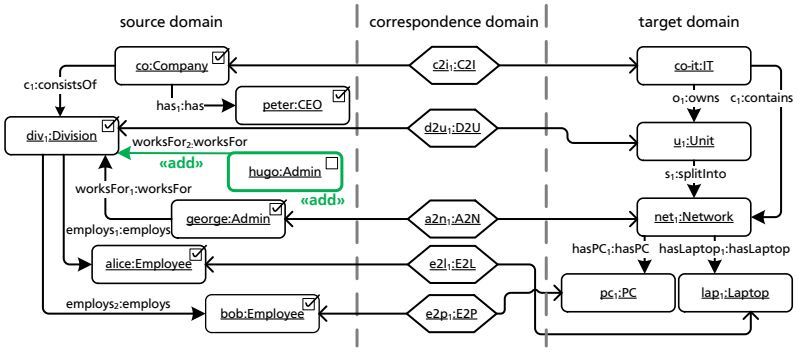


Figure 40: Scenario 4: Hire a new `Admin`

The important point regarding all scenarios and especially the first two scenarios is that the model synchronization algorithm shall retain as much information as possible. Considering the Scenarios 1 and 2 where an `Employee` is dismissed or hired, this model change has definitely no influence on other `Employees`. Hence, it has to be ensured that the model synchronization retains the additional information which has been placed when selecting to either grant an `Employee` a `PC` or a `Laptop`.

Regarding the synchronization task it can be seen easily that our batch algorithm (cf. Chap. 7) would be able to "synchronize" the models, as it would accept the changed source model and apply a batch transformation on it. Afterwards, consistency is guaranteed but information has been lost or had to be re-introduced during the batch transformation process.

Overall, using an additional diff and calculating (potentially) affected elements instead of batch transforming the changed model is an important aspect for two reasons:

- Information of unaffected elements can be retained that otherwise has to be re-introduced to the model integration.
- The larger a model gets the more efficient it will be to consider just a sufficiently small sub-model for propagating the changes and recover a consistent model integration.

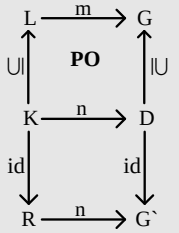
9.2 EXTENDED OPERATIONAL RULE DERIVATION

Synchronizing models means to propagate changes of one domain to another. As described previously, such changes could be the modification of an element (not supported yet), the addition of an element, or the deletion of an element. The propagation towards the opposite domain will be controlled by an appropriate control algorithm that traverses the input model considering the changes and invokes specific actions to update the opposite model. The actions triggered are again the operational rules derived from the declarative rule specification. The addition of elements can be handled by applying a *forward rule* as presented in Def. 13 in Chap. 3. Therefore, a newly added element will be treated as an untransformed element and hence can be transformed in a straight-forward manner. Handling a deleted element can obviously not be handled by transforming this element with a forward rule as this element has already been transformed and was deleted recently. In this case, a different operational rule has to be applied that *revokes* the results of the previously applied forward rule: all elements that have been created in the opposite domain by applying the forward rule have to be removed. Thus, the element to which this rule was applied has to be marked as untransformed. Such an operation is called an *inverse forward rule*.[3]

As the derivation of source and forward rules is insufficient for incremental change propagation, we formally introduced in [LAVS12b] the derivation of so-called *inverse forward rules*. Such rules invert the result of a previous transformation (i.e., rule application) by *untransforming* previously transformed elements as they remove the corresponding elements from the graph triple. As inverse forward rules only delete elements, we define monotonic deleting rules first:

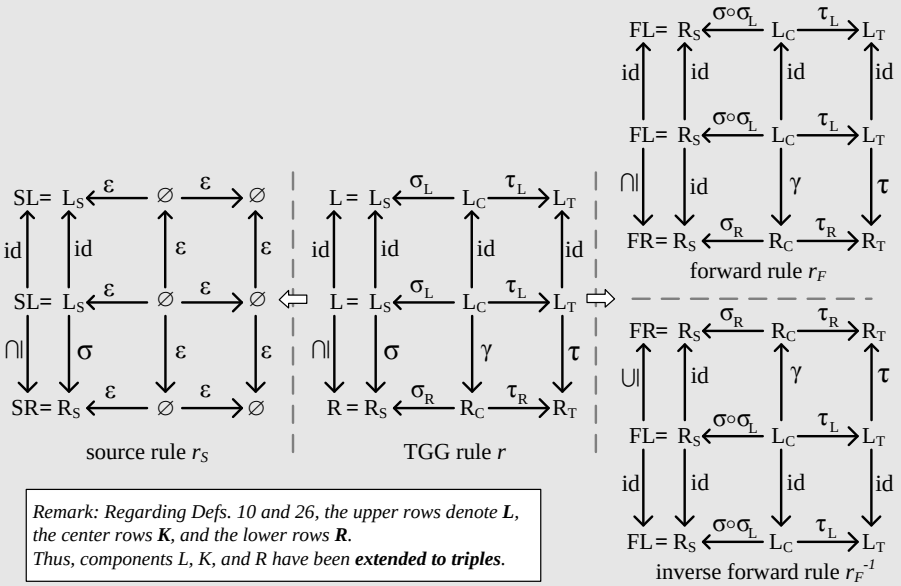
3 Of course it is possible to derive additional operational rules for propagating changes. Such a rule would instead of creating or deleting elements regard the existing elements and update (override) specific attribute values with their updated values.

Definition 26 (Monotonic Deleting Rules)
A monotonic deleting rule $r := (L, K = R)$, *is a pair of typed graph triples s.t.* $L \supseteq R$. *A rule* r rewrites *(via deleting elements) a graph triple* G *into a graph triple* G' *via a match* $m : L \to G$, *denoted as* $G \overset{r@m}{\rightsquigarrow} G'$, *iff* $n : R \to G'$ *can be defined by building the* pushout complement $H = G'$ *as denoted in the diagram.*



The elements in $L \setminus R$ of a monotonic deleting rule are referred to as *deleted elements*. Using this definition, the extended operational rule derivation is formalized as follows, where the derivation of the source and forward rule remain as-are compared to Def. 13) in Chap. 3.

Definition 27 ((Extended) Derived Operational Rules)
Given a $TGG = (TG, \mathcal{R})$ *and a rule* $r = (L, R) \in \mathcal{R}$, *a* source rule $r_S = (SL, SR)$, *a* forward rule $r_F = (FL, FR)$ *and an* inverse forward *rule* $r_F^{-1} = (FR, FL)$ *are derived as follows:*



The forward rule r_F *can be applied according to Def. 10, i.e., this involves building a pushout to* create *the required elements, while the inverse forward rule* r_F^{-1} *involves building a pushout complement to* delete *the required*

elements according to Def. 26. Given a forward rule r_F, the existence of rule r_F^{-1}, which reverses an application of r_F up to isomorphism, can be shown according to Fact 3.3 in [EEPT06].

Again, a control algorithm has to keep track of which source elements are transformed by a rule application. Hence, we equip in concrete syntax every *transformed* element with a *checked box*, and every *untransformed* element with an *unchecked box*.

Example

From Rule (c) (Fig. 36), the operational rules r_S, r_F, and r_F^{-1} depicted in Fig. 41 are derived. The source rule extends the source graph by adding an `Admin` to an existing `Division`, while the forward rule r_F *transforms* (denoted as $\square \to \boxtimes$) an existing `Admin` by *creating* a new `A2N` link and `Network` in the corresponding `IT`. The inverse forward rule *untransforms* (denoted as $\boxtimes \to \square$) an `Admin` in a `Division` by *deleting* the corresponding link and `Network`, i.e., revoking the modifications of the forward rule. In addition to the already introduced merged representation of `L` and `R` of a rule, we further indicate deleted elements by a "––" markup and red color. Forward and inverse forward rules match the same context element and retain the checked box (denoted as $\boxtimes \to \boxtimes$).

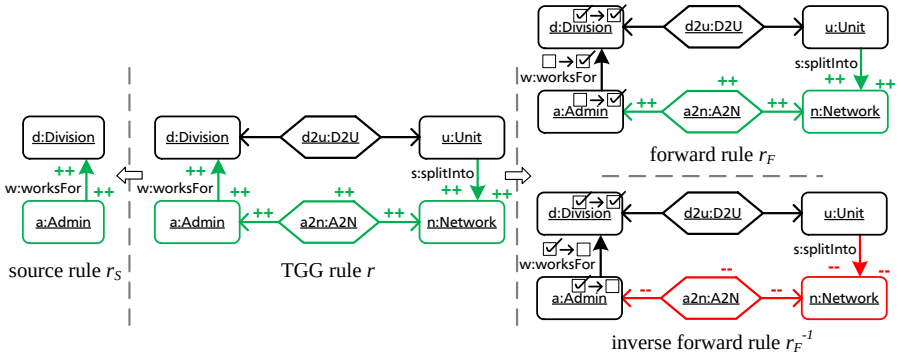


Figure 41: Extended source and forward rules derived from Rule (c)

9.3 PRECEDENCE PRESERVING GRAPH TRIPLES

To propagate changes in the order determined by the precedence analysis presented in Chap. 6, we have to ensure that this order is always sufficient for the control algorithm to determine a traversal order.

The problem is that the approach of this thesis utilizes one domain only to build the mentioned sorting. As the incremental algorithm will have to delete elements in the opposite domain (unchanged

model) in order to propagate a change of the input domain (changed model), it may happen that elements in the opposite domain are removed which would have been required as context elements for another change propagation. As the precedence graph of the input domain induces a partial order as well as the precedence graph of opposite domain, we have to require that these partial orders do not contradict each other but induce the same precedences between cross-domain connected elements. Formally, this fact is defined as follows for a forward synchronization:

Definition 28 (Forward Precedence Preserving Graph Triples)
Given a graph triple $G = G_S \xleftarrow{h_S} G_C \xrightarrow{h_T} G_T$ *and two corresponding precedence graphs* $\mathcal{PG}_S$ *and* $\mathcal{PG}_T$*. We require that at least one element of each equivalence class is connected to the correspondence domain:*
$\forall\, EQ_S \in V_{\mathcal{PG}_S} \exists\, n \in V_{G_C} : h_S(n) \in EQ_S \wedge$
$\forall\, EQ_T \in V_{\mathcal{PG}_T} \exists\, n \in V_{G_C} : h_T(n) \in EQ_T.$
With such connected triples we define cross-domain connectivity *as follows:*
For $EQ_S \in V_{\mathcal{PG}_S}$ *and* $EQ_T \in V_{\mathcal{PG}_T}$*, the predicate* cross-domain-connected *on pairs of equivalence classes in precedence graphs of different domains is defined as follows: cross-domain-connected*$(EQ_S, EQ_T) :=$
$\exists\, n_C \in V_{G_C}$ *s.t.* $h_S(n_C) \in EQ_S \wedge h_T(n_C) \in EQ_T$.
Given $EQ_S, EQ'_S \in V_{\mathcal{PG}_S}, EQ_S \neq EQ'_S$ *and* $EQ_T, EQ'_T \in V_{\mathcal{PG}_T}, EQ_T \neq EQ'_T$ *s.t. cross-domain-connected*$(EQ_S, EQ_T) \wedge$ *cross-domain-connected*(EQ'_S, EQ'_T)*. The graph triple* G *is* forward precedence preserving *iff*
$\exists$ *path* $p_T(EQ_T, EQ'_T) = EQ_T \cdot e_{T_1}^{\alpha_{T_1}} \cdot \ldots \cdot e_{T_n}^{\alpha_{T_n}} \cdot EQ'_T$ *s.t.* $\alpha_{T_i} = + \,\forall\, i \in \{1, \ldots, n\}$
$\Rightarrow$
$\exists$ *path* $p_S(EQ_S, EQ'_S) = EQ_S \cdot e_{S_1}^{\alpha_{S_1}} \cdot \ldots \cdot e_{S_n}^{\alpha_{S_n}} \cdot EQ'_S$ *s.t.* $\alpha_{S_i} = + \,\forall\, i \in \{1, \ldots, n\}$

To achieve precedence preservation for both forward and backward synchronization, the graph triples are also required to be backward precedence preserving (which is defined analogously).

Example

All four scenarios (cf. description of Figs. 37, 38, 39, and 40) satisfy this property as in each scenario the precedence graph for source domain and the precedence graph for target domain domain do not induce any contradictory precedences.

Remark: Up to now, we have not been able to construct a practice relevant and not theoretical TGG and appropriate graph triple that violates the requirement induced by Def. 28. From a formal point

of view, a TGG fulfills this restriction whenever every TGG rule creates at least one element in the correspondence domain that connects created elements of the source and target domain. Unfortunately, we lack empirical information if this is an actual restriction.

10 INCREMENTAL CONTROL ALGORITHM

Incremental bidirectional model synchronization with TGGs is realized with a control algorithm that accepts a triple $G = G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$, an update graph triple [HEO+11] for the source domain $\Delta_S = G_S \leftarrow D \rightarrow G'_S$, the pre-compiled precedence function for the source domain $\mathcal{PF}_S$, and the precedence graph $\mathcal{PG}_S$ used in a previous batch or incremental transformation. This algorithm returns a consistent graph triple $G' = G'_S \leftarrow G'_C \rightarrow G'_T$ with all changes propagated to the correspondence and target domain. Figure 42 depicts this process: A graph triple G and its changes in the source domain Δ_S are passed to and processed by the control algorithm. A completed triple G' is returned (precedence function and graph are not displayed).

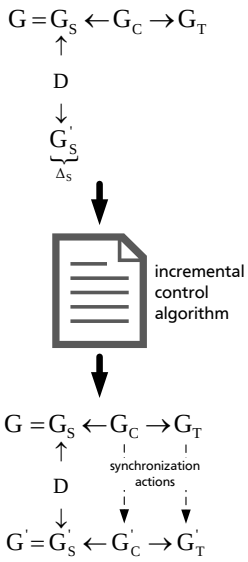


Figure 42: Schematic incremental synchronization process

10.1 THE CONTROL ALGORITHM

The algorithm presented in this chapter basically runs in three different phases:

Phase (i): Untransform dependencies of deletions and propagate deletions

Phase (ii): Untransform dependencies of additions

Phase (iii): Transform all additions and untransformed elements

In phase (iii), this algorithm utilizes the previously introduced precedence-driven batch algorithm (Algorithm 1, cf. Chap. 7). Regarding the processing order, the algorithm has to find a way to delete elements in the opposite domain without compromising the transformation of existing elements. As a (formal) restriction, edges can only be added (deleted) together with adjacent nodes, hence we focus on nodes only. In practice, Ecore for example assigns all edges to nodes, which overcomes this restriction.

Algorithm 2 Incremental Precedence TGG Algorithm

```
1: procedure propagateChanges(G, Δ_S, PF_S, PG_S)
2:   for (node n⁻ ∈ Δ⁻) do
3:     untransform(n⁻, PG_S)
4:   end for
5:   (G_S⁻, PG_S⁻) ← remove all n⁻ in Δ⁻ from G_S and update PG_S
6:   (G_S⁺, PG_S⁺) ← insert all n⁺ in Δ⁺ to G_S⁻ and update PG_S⁻
7:   if PG_S⁺ is cyclic then
8:     terminate with exception ▷ Additions invalidated G_S⁺ (i.e., G'_S)
9:   end if
10:  for (node n⁺ ∈ Δ⁺) do
11:    untransform(n⁺, PG_S⁺)
12:  end for ▷ At this point G has changed to G* = G'_S ← G*_C → G*_T
13:  return (G'_S ← G'_C → G'_T) ← transform(G*, PF_S) ▷ Call Algo. 1
14: end procedure

15: procedure untransform(n, PG_S)
16:   deps ← all nodes in all equiv. classes in PG_S
                with incoming edges from EQ(n)
17:   for node dep in deps do
18:     if dep is transformed then
19:       untransform(dep, PG_S)
20:     end if
21:   end for
22:   neighbors ← all nodes in EQ(n)
23:   for node neighbor in neighbors do
24:     if n is transformed then
25:       applyInverseRule(n) ▷ Throw exception if Def. 28 is violated
26:     end if
27:   end for
28: end procedure
```

Procedure propagateChanges requires the original graph triple G, the set of changes Δ_S represented as a graph triple as well,[1] the pre-

1 To represent the changes in form of a graph triple is necessary to define the gluing between the original graph and its changes. From a practical point of view, the subsets Δ^- and Δ^+ can be derived easily which will contain nodes only that have been deleted or added.

calculated precedence function $\mathcal{PF}_S$ and the precedence graph $\mathcal{PG}_S$ from a previous transformation run.

Phase (i) (lines (2)–(5)): In a first step (lines 2–4), all deletions are propagated towards the opposite domain. For this reason, procedure UNTRANSFORM is called on line (3) for every deletion in Δ^-. In this procedure (see detailed description below), all elements that depend on a deleted element are processed and their earlier transformation is revoked. The traversal order is determined due to the precedence graph. After having removed all elements in the opposite domain that were created due to the (direct or indirect) existence of now deleted source element, the algorithm actually removes all deletions on line (5) from the source domain graph G_S as well as from the precedence graph $\mathcal{PG}_S$.

Phase (ii) (lines (6)–(12)): The second phase starts on line (6) by introducing all newly added elements Δ^+ to the previously cleared source domain graph G_S^- and the precedence graph $\mathcal{PG}_S^-$. Note that at this point, the updated source domain graph G_S^+ is equal to the updated source domain graph G'_S. If the precedence graph becomes cyclic via these additions, the algorithm terminates with an exception on line (8) as no transformation will be possible. Additions may produce new dependencies between elements. Such a new dependency states that a previously existing and already transformed element could now also be transformed by using the added element as (direct or indirect) context. As all possible graph triples should be derivable (completeness property), these dependent elements have to be untransformed in the same manner as for the dependent elements of deletions (cf. phase (i)). For this reason, procedure UNTRANSFORM is called for any addition on line (11). When the for-loop on lines (10)–(12) terminates, the graph triple consists of G'_S which is the source domain graph with all applied changes and the correspondence and target domain graphs which are partially integrated with the source domain graph. The triple itself denotes an intermediate state of a batch transformation run where G'_S would have been used as the input graph. This means that various elements in G'_S exist that are untransformed by now.

Phase (iii) (line (13)): The third phase passes now the graph triple and the precedence function to the batch algorithm (Algorithm 1) of Chap. 7. This algorithm will process all untransformed elements in the presented manner and, therefore, complete the update process by transforming all additions and such elements that have be untransformed due to their dependency to a model change.

Procedure UNTRANSFORM$(n, \mathcal{PG}_S)$ (lines (15)–(28)): This procedure is responsible for revoking previous transformations of elements in an appropriate order such that all dependencies are regarded. Firstly, function $\mathrm{EQ}(n)$ is used to determine all dependent nodes that (potentially) used node n in a previous transformation run as an con-

text element.[2] Hence, all these nodes have to be untransformed first, which is achieved via calling method UNTRANSFORM recursively on line (19) for all dependent nodes. Reaching the recursion end of this dependency path, all nodes within the equivalence class are untransformed by applying the inverse forward rule (cf. Def. 27) of the previously applied forward rule on line (25).[3] Two things are important at this point: First, all elements within an equivalence class have to be untransformed as our understanding of precedences states that pairings of nodes within an equivalence class can be transformed in one single transformation step. As the decision, which combination will be transformed together will be made during the actual pattern matching process of applying a rule, all possible combinations should be available in order to retain the completeness property. The second important aspect at this point is that if the application of the inverse forward rule fails, we know that a specific context element in the target domain does not exist and, therefore, it must have been removed in previous inverse rule application. Hence, the graph triple was not precedence preserving and Def. 28 has been violated.

Example

We are now testing our recently presented incremental control algorithm with the four scenarios presented in Sect. 9.1.

Scenario 1: Regarding our first incremental scneario (cf. Fig. 37), the algorithm has to propagate the deletion of a single `Employee` object. Hence, the algorithm receives the graph triple G as depicted in Fig. 35 plus the changes stored in Δ_S, with the subsets $\Delta^- = \texttt{bob}$ and $\Delta^+ = \emptyset$.

The precedence graph for the source domain of G is depicted in Fig. 43.

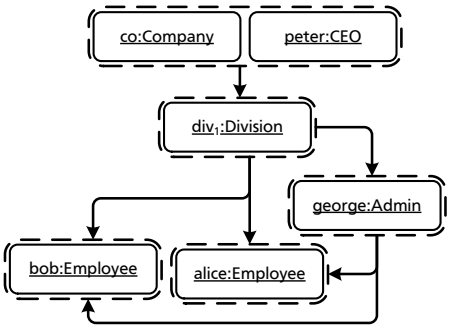


Figure 43: Scenario 1: Precedence graph $\mathcal{PG}_S$ for the source domain of triple G (cf. Fig. 35)

2 EQ(x) returns the appropriate equivalence class of node x.

3 Recently, we expect that we keep book of which rules have been applied in order to exactly revert the previous rule application. Nevertheless, we discuss within the evaluation chapter of this part (cf. Chap. 11.), ideas how to avoid this additional overhead.

In phase (i), the algorithm starts untransforming all deletions on line (2). As node `bob` is the only element in Δ^-, procedure UNTRANSFORM is called only once on line (3). According to the precedence graph $\mathcal{PG}_S$ (cf. Fig. 43), node `bob` is not directly or indirectly dependent of any other nodes, hence set `deps` remains empty on line (16). Furthermore, the equivalence class containing node `bob` consists of this node only, and, therefore, the same is true for the set `neighbors` (cf. line (22)). As `bob` is transformed (☑) the inverse forward rule is applied. Calling APPLYINVERSERULE untransforms `bob` by applying the inverse forward rule of Rule (d). Note that with an appropriate bookkeeping data structure this method is aware of all previous rule applications and applies the correct inverse forward rule to the same match used previously by the forward transformation. In this case, the target domain element $\texttt{pc}_1$ is deleted together with the appropriate correspondence link. Returning from procedure UNTRANSFORM, the source domain graph G_S and the precedence graph $\mathcal{PG}_S$ are updated (depicted in Figs. 44 and 45).

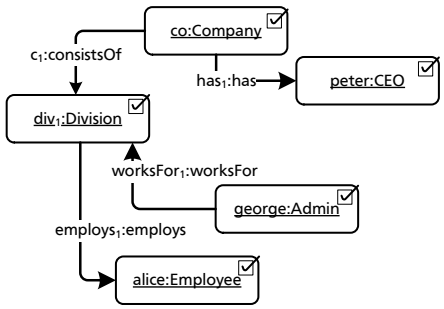


Figure 44: Scenario 1: Updated source domain graph G_S^+ after line (6)

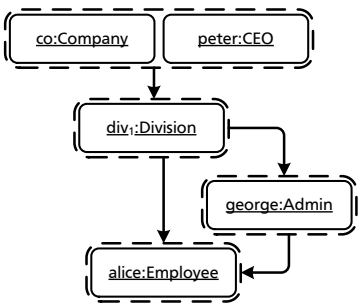


Figure 45: Scenario 1: Updated precedence graph $\mathcal{PG}_S^+$ after line (6)

Therefore, all deletions are actually removed on line (5). Phase (ii): As no additions exist, line (6) does not alter G_S or $\mathcal{PG}_S$ any further. The updated precedence graph contains no cycles and the algorithm continues clearing dependencies of additions. As no elements have been added, the loop on lines (10)–(12) is skipped. Finally in phase (iii), the altered graph triple is passed to the batch transformation control algorithm of Chap. 7. As all elements are already transformed,

the algorithm returns the same graph triple, i.e., the one depicted in Fig. 46.

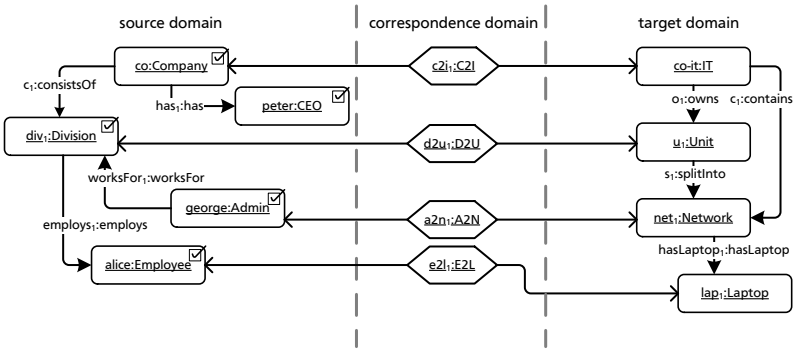


Figure 46: Scenario 1: Consistently updated graph triple G' after line (13)

Scenario 2: The algorithm has to handle the addition of an `Employee`. Therefore, the algorithm skips phase (i) of untransforming elements that have been deleted (cf. lines (2)–(5)) as a new `Employee` has been added only. In phase (ii), the algorithm starts adding this element to G_S and the precedence graph on line (6). The updated precedence graph $\mathcal{PG}_S^+$ contains no cycles, as the additional `Employee` introduces a leaf to the precedence graph only. Moreover, no dependent elements are determined by the called method UNTRANSFORM on lines (16)–(21). Thus, this `Employee` has no neighbors since `Employee` elements are not processed together with other elements (cf. Rules (d) and (e) in Fig. 36). Therefore, lines (22)–(28) do not apply any change to the graph triple. Returning from UNTRANSFORM on line (11) the for-loop terminates as a single element was added only and passes the graph triple G^* to the batch algorithm. In phase (iii), the batch algorithm has to process only the newly added `Employee` while all other elements in G^* remain untouched. As our TGG is non-functional, the batch algorithm may choose from two rules to be applied to the added `Employee` and, hence, the depicted final graph triple in Fig. 47 denotes one of the possible consistent graph triples that could be produced by our algorithm.

Scenario 3: In this incremental scenario, the update procedure will involve some more actions, as the deletion and the addition of a `CEO` have to be handled (cf. Fig. 39). Regarding phase (i), the set Δ^- consists of node `peter`, and, therefore, procedure UNTRANSFORM is called on line (3) to revoke the effects of those forward rules that had been applied to this node and its dependent elements. Considering the precedence graph $\mathcal{PG}_S$ depicted in Fig. 43, the set `deps` consists of the element $\texttt{div}_1$ after line (16). For this reason, the loop starting on line (17) calls procedure UNTRANSFORM recursively to clear the dependent integration of this element. The recursion stack is depicted graphically in Fig. 48.

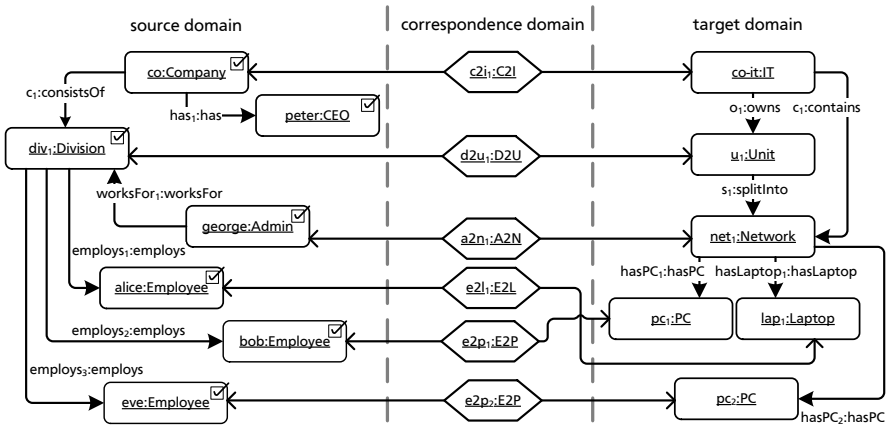


Figure 47: Scenario 2: Consistently updated graph triple G′ after line (13)

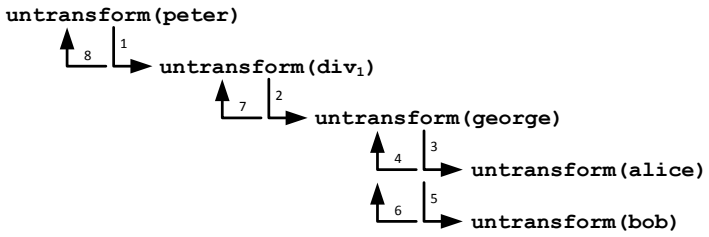


Figure 48: Scenario 3: Recursion stack for revoking previous transformations of dependent elements of the `CEO peter`

The recursion reveals that nodes `george`, `alice`, and `bob` are in `dep` (line (16)) and, thus, they have to be untransformed. Again, recursion for these elements is started, where UNTRANSFORM for node `george` is called first (cf. arrow 2 of Fig. 48). This node has the two dependencies `alice` and `bob`, calling recursively UNTRANSFORM for these elements (cf. arrows 3–6 of Fig. 48). The recursion reaches its end as neither of these elements have dependents and, consequently, the appropriate inverse rules are applied to all elements in their equivalence classes (which are the nodes themselves only). Now, node `george` can be untransformed by applying the inverse of Rule (c) on line (25). Returning from the recursion (cf. arrow 7 of Fig. 48), procedure UNTRANSFORM should be called for nodes `alice` and `bob` again (cf. arrows 8–10 of Fig. 48) as they are still in the set `dep`. As these dependencies are already untransformed, the if-clause on line (18) prevents the algorithm from repeating unnecessary recursion steps. At this point the algorithm has definitely untransformed all dependencies below any of these two dependent elements (if there were such elements). Now, element $\texttt{div}_1$ is untransformed by applying the inverse forward rule of Rule (b) before this recursion returns to the topmost level, where the equivalence class containing `co` and `peter` is completely untransformed on lines (22)–(27). Updating the source domain graph G_S and the appropriate precedence graph $\mathcal{PG}_S$ removes the element `peter`. In

phase (ii) starting on line (6), the new `CEO` named `will` is added to the precedence graph and the source domain graph. As the precedence graph is acyclic, the algorithm starts untransforming all dependencies of the newly added element.[4] As all dependencies (i.e., $\mathtt{div}_1$) are already untransformed, the for-loop on lines (10)–(12) has nothing to do. The algorithm has so far untransformed every single element as a consequence of the applied changes to the source domain graph. This happened because an element has been removed (i.e., `peter`) which was transformed together with the root node `co` and, therefore, all further elements depended on this first transformation step. In phase (iii), the triple consisting of a completely untransformed G'_S (G^*_C and G^*_T are empty) is passed on line (13) to the batch algorithm and the final result is depicted in Fig. 49 (note that we assume that the algorithm increased all running numbers by one).

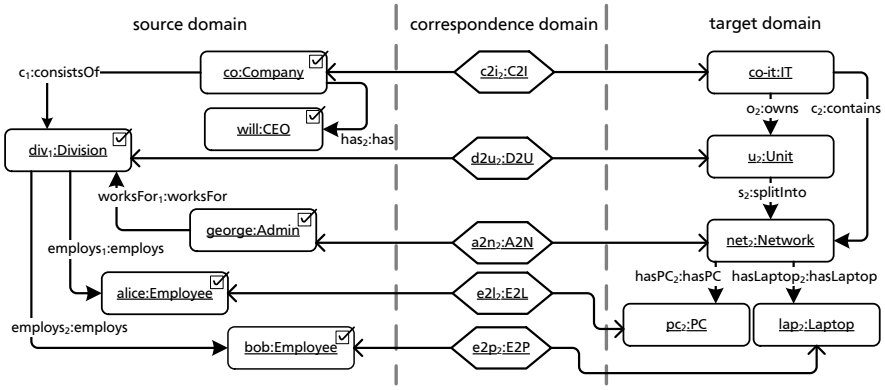


Figure 49: Scenario 3: Consistently updated graph triple G' after line (13)

Scenario 4: In the fourth and last incremental scenario, only one change was applied to the source domain graph G_S: a new `Admin` named `hugo` has been hired. As no deletions have to be handled, the algorithm has nothing to do in phase (i). In phase (ii), the addition has to be regarded. From a semantical point of view, employing an additional `Admin` allows for the maintenance of another `Network` and, furthermore, the `Employees` may be distributed in these `Networks`. From a formal point of view, Rules (d) and (e) require an `Admin` object as context (cf. Fig. 36). As the model has now two `Admins`, applications of Rules (d) or (e) may freely choose between these two `Admins`. For this reason, it is necessary that both `Employees` are untransformed first, and afterwards the final batch transformation step (phase iii) may again freely choose which `Admin` will be used as context. This is exactly, how the algorithm behaves. Iterating through all deletions has no effects on the graph triple, as no deletions exist in this case. Introducing node `hugo` to G^+_S and $\mathcal{PG}^+_S$ on line (6) returns newly added dependencies depicted in Fig. 50.

4 The fact that this is an important step in the algorithm will become clear in the discussion of the synchronization process for scenario 4.

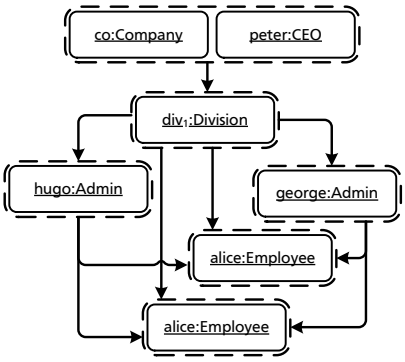


Figure 50: Scenario 3: Updated precedence graph $\mathcal{PG}_S^+$ after line (6)

Looping through all additions on lines (10)–(12) untransforms elements `alice` and `bob`. The cleared graph triple contains three untransformed nodes (□) depicted in Fig. 51.

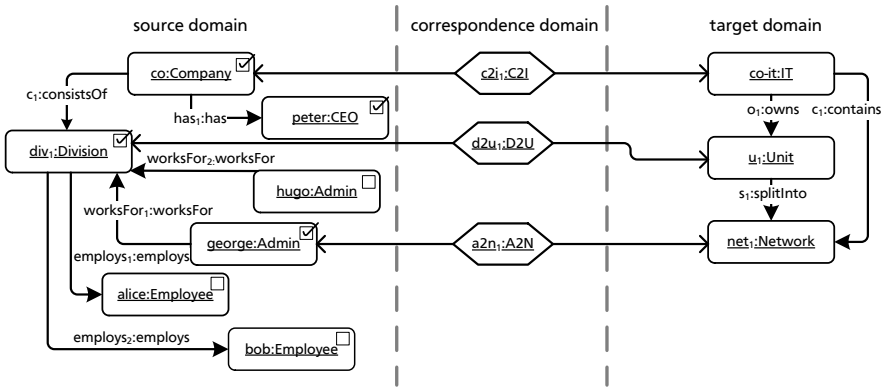


Figure 51: Scenario 4: Updated graph triple G* after line (12)

Finally in phase (iii), the batch transformation algorithm called on line (13) will now transform node `hugo` first, as all dependencies are successfully processed already (☑). Afterwards, handling elements `alice` and `bob` in any order allows for the free choice of selecting any of the two existing `Networks` in the target domain. One possible final result is depicted in Fig. 52.

10.2 UPDATING A PRECEDENCE GRAPH $\mathcal{PG}$

Considering lines (5) and (6) of Algorithm 2 the precedence graph $\mathcal{PG}_S$ has to be updated reflecting the actually applied changes. Hence, it has to be discussed how this update can be achieved efficiently and completely such that the updated precedence graph is identical compared to a precedence graph that would have been derived directly from the updated input graph. This is necessary in order to ensure

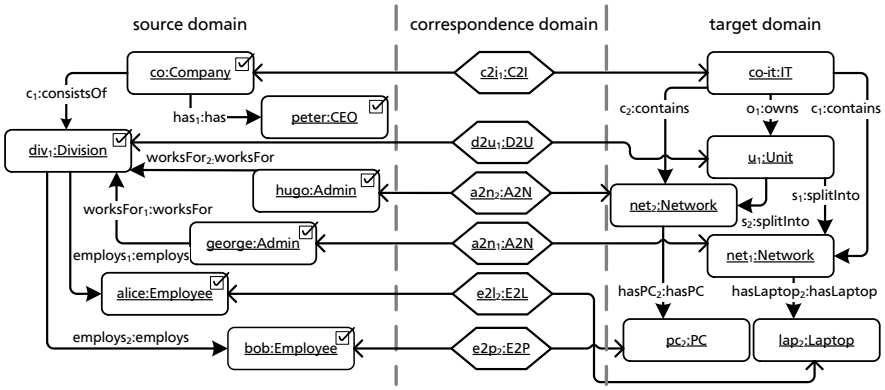


Figure 52: Scenario 4: Consistently updated graph triple G' after line (13)

that further modifications can be propagated appropriately without compromising correctness and completeness of our algorithm.

Updating a precedence graph (i.e., adding or removing a node n) has to regard all edges that exist only due to the existence of n and the equivalence class of n. Due to the definition of $\mathcal{PG}_S$ (cf. Def. 25), we know that edges from and to the equivalence class of n exist only in $\mathcal{PG}_S$ if n is part of a path that expresses a certain precedence relationship. All situations, that may lead to such an edge are systematically described in the following. Hence, we argue which sets of elements have to be considered in order to regard *all* effects due to the change of n. Edges of the precedence graph may depend on the existence of n for the following three reasons:

(i) All edges that express a direct context dependency on n (n is direct context of y, i.e., $(n, y) \in \lessdot_S$). All these elements are subsumed in a set which is denoted as *direct-dependent(n)*.

(ii) All edges that express an inverse direct context dependency on n (x is direct context of n, i.e., $(x, n) \in \lessdot_S$). All these elements are subsumed in a set which is denoted as *direct-context (n)*.

(iii) All edges that express a direct context or equivalence dependency between nodes x and y, where n is an element of the path between x and y. Here we have to distinguish four further reasons:

(a) A path $x \cdot \ldots \cdot n \cdot \ldots \cdot y$ exists, such that $(x, y) \in \doteq_S$ (a rule may use n as context to create x, y). In this case x and y are already in the set direct-dependent(n).

(b) A path $x \cdot \ldots \cdot n \cdot \ldots \cdot y$ exists, such that $(x, y) \in \doteq_S$ and $(x, n), (n, y) \in \doteq_S$ (a rule creates x, y and n together). In this case n, x and y are in the same equivalence class.

(c) A path $x \cdot \ldots \cdot n \cdot \ldots \cdot y$ exists, such that $(x, y) \in \lessdot_S$ and $(n, y) \in \doteq_S$ and $(x, n) \in \lessdot_S$ (a rule creates n and y together while

using x as context). In this case is x in the set direct-context(n) and y in the same equivalence class of n.

(d) A path $x \cdot \ldots \cdot n \cdot \ldots \cdot y$ exists, such that $(x, y) \in \lessdot_S$ and $(n, y) \in \lessdot_S$ (a rule creates y while using x, n as context). In this case is y in the set direct-dependent(n) and x in the set direct-context(y).

All other edges in the precedence graph are independent of n as these edges have not been added due to the existence of n; as argued above.

As described above, different sets of elements can be distinguished that have to be regarded due the change of n. Therefore, updating a precedence graph requires to regard the sets direct-dependent(n), direct-context(n), and direct-context(direct-dependent(n)) as well as all nodes in the same equivalence class of n. These sets are graphically depicted in Fig. 53. As these sets of direct dependent and direct context elements form a circle around a model change, we denote this overall set a n_{circle}.

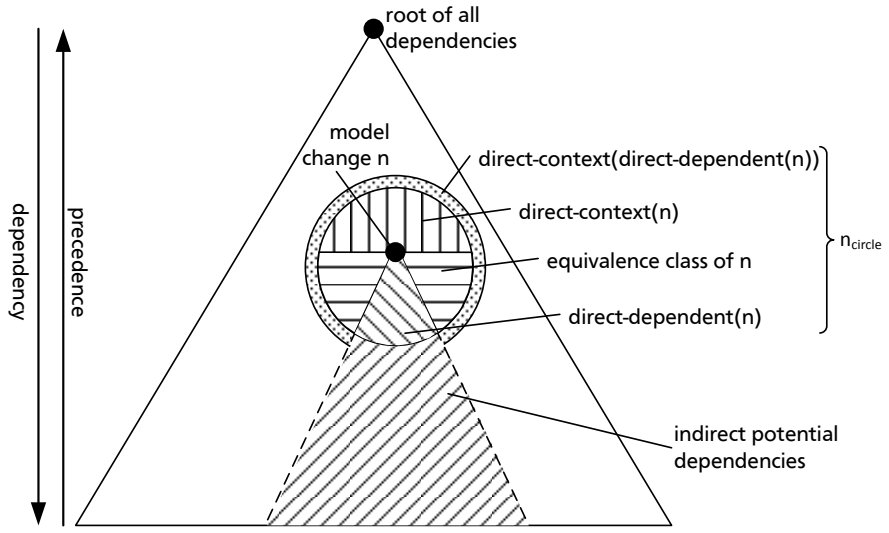


Figure 53: Different sets of affected elements to be regarded while updating a precedence graph

Finally, regarding the runtime complexity of updating a precedence graph, we know that paths can be at most of length k, where k denotes the maximum rule size of all rules. Therefore, updating a precedence graph has a $\mathcal{O}((n_{circle})^k)$ complexity.

10.3 FORMAL PROPERTIES

In this section, we prove that our algorithm retains all formal properties introduced by [Sch95, SK08] (cf. Sect. 4.3).

Theorem 2
Algorithm 2 is correct, complete and efficient for any local complete TGG and any corresponding graph triple that is forward precedence preserving (Def. 28).

Proof
Correctness:
Lines (2)–(12) of the algorithm only invert previous rule applications. The order of rule applications is directed by the precedence graph (Def. 25), which represents potential dependencies between nodes, i.e., a node x is a dependency of another node y, which may be transformed by applying a rule that matches x as context. These are potential dependencies as actual rule applications may select other nodes instead of x. Nevertheless, y potentially depends on x. The algorithm traverses all dependencies of every deleted/added node and applies the inverse of the rule used in a previous transformation. Demanding precedence preserving graph triples (Def. 28) guarantees that $\mathcal{PG}_S$ is sufficient to correctly revoke forward rules in a valid order. If an element on the target side is deleted by applying an inverse forward rule,[5] which is still in use as context for another element in the target graph, we know that the forward precedence preserving property is violated. This also guarantees that deleting elements via building a pushout complement (Def. 26) is always possible and cannot be blocked due to "dangling" edges. In combination with bookkeeping of previously used matches, it is guaranteed (Def. 13) that the resulting triple is in the same state as it was before transforming the untransformed node.

It directly follows that if the triple G was consistent, the remaining integrated part of G remains consistent. Since UNTRANSFORM inverts rule applications of a previous transformation, we know that the graph triple after line (12) is a correct intermediate graph triple produced by the batch transformation algorithm. As shown in Sect. 7.2, the precedence-driven TGG batch algorithm is correct, so Algorithm 2 is also correct. If the application of forward or inverse forward rules fails, the algorithm returns an appropriate exception (e.g., the model change caused schema incompliance). □

Completeness:
The correctness proof shows that the incremental update produces a triple via a sequence of rule applications that the batch algorithm could have chosen for a forward transformation of G'_S. In cases where the input violates restrictions of the algorithm (e.g., the requirement for local complete TGGs), an appropriate exception will be returned.

5 Note that an appropriate bookkeeping data structure knows which rule has been applied.

Completeness arguments from Sect. 7.2 for the batch algorithm can, hence, be transferred to this algorithm. □

Efficiency:
Efficiency is influenced mainly by the cost of (i) untransforming dependent elements of a deleted or added node (lines (2)–(4) and (10)–(12)), (ii) updating the precedence graph and the graph triple itself (lines (5) and (6)), and (iii) transforming all untransformed elements via our precedence-driven TGG batch algorithm (line (13)). The number of deleted/added nodes ($|\Delta^-| + |\Delta^+|$) and their dependencies is denoted by $n^{\text{trans}}_{\lessdot,\doteq}$ (i.e., all elements with a (transitive) precedence of $\lessdot$ or $\doteq$) as depicted in Fig. 54.

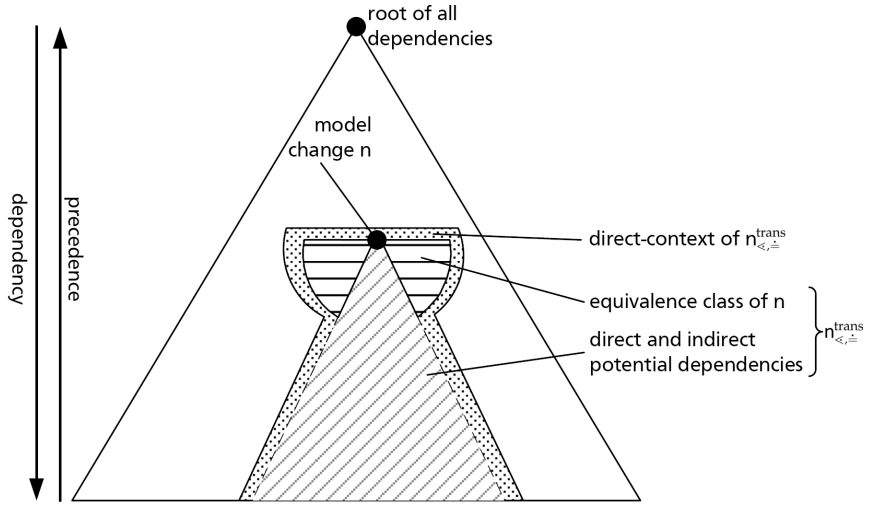


Figure 54: Regarded sets of nodes during a synchronization run (without updating $\mathcal{PG}_S$)

In procedure UNTRANSFORM, a recursive depth-first search on the precedence graph $\mathcal{PG}_S$ is invoked starting at a certain node. Depth-first search has a worst-case complexity of $\mathcal{O}((n^{\text{trans}}_{\lessdot,\doteq})^k)$. If the algorithm encounters an already untransformed element on line (18), we know for sure that all subsequent elements are already untransformed and, therefore, can safely terminate recursion. Independent of the position of the changed element, UNTRANSFORM traverses each dependent element exactly once. Finally, applying the inverse operational rule (line (25)) is (at most) of the same complexity as the appropriate previous rule application since the rule and match are already known (due to the bookkeeping when a rule is applied). Considering phases (i) and (ii), we know that $n^{\text{trans}}_{\lessdot,\doteq}$ elements are untransformed, and that every element is treated exactly once.
Updating G_S on line (5) (respectively line (6)) involves deleting (inserting) $|\Delta^-|$ ($|\Delta^+|$) elements $m \in \Delta^-(\Delta^+)$ and involves updating a number of adjacent nodes (degree(m)) for each m. We assume that actions such as deleting or adding elements in graphs as well

as type and existence checking can be performed in constant steps, i.e. $\mathcal{O}(1)$ Thus, the complexity of updating G_S on line (5) and (6) can be estimated with $\mathcal{O}(|\Delta_S|)$, as Δ_S contains all nodes and edges that have been changed and, therefore, need to be revised. Additionally, updating $\mathcal{PG}_S$ has a complexity of $\mathcal{O}((n_{\text{circle}})^k)$ (cf. Sect. 10.2). Finally, transforming the rest of the prepared graph (line (13)) has $\mathcal{O}((n^{\text{trans}}_{\preccurlyeq,\doteq} + \text{direct-context}(n^{\text{trans}}_{\preccurlyeq,\doteq}))^k)$ complexity (cf. proof efficiency of Theorem 1): Only added elements, their dependencies, and the dependencies of removed elements have been untransformed, $n^{\text{trans}}_{\preccurlyeq,\doteq}$ refers only to these elements, and not to all elements in G_S. Thus, as argued in Sect. 10.2 the additional effort of updating the precedence graph has a complexity of $\mathcal{O}((n_{\texttt{circle}})^k)$. This set only adds direct context elements of all changes and transitively dependent elements as all direct dependent and equivalent elements are already included in $n^{\text{trans}}_{\preccurlyeq,\doteq}$. Therefore the algorithm is polynomial in the number of changes, their dependencies and their context elements ($n_\delta = |\Delta^-| + |\Delta^+| + n^{\text{trans}}_{\preccurlyeq,\doteq} + \text{direct-context}(n^{\text{trans}}_{\preccurlyeq,\doteq}) + n_{\text{circle}}$) and not in the size of the graph triple: $n_\delta \leqslant n$. □

Remark: In practice, a complete synchronization algorithm would also have to update the precedence graph of the target domain $\mathcal{PG}_T$. This, of course, induces additional runtime costs that have a similar complexity compared to the costs for updating $\mathcal{PG}_S$. Nevertheless, w.l.o.g. this combined complexity remains still in the same complexity class as the complexity does not change if the size of the regarded set is changed.

Again, as TGGs are symmetric [HEO+11], all arguments can be transferred analogously to backward transformations.

11

EVALUATION OF THE INCREMENTAL ALGORITHM

In Part iii, we presented Algorithm 2 that uses the results of the static precedence analysis and the extended operational rule derivation to solve the task of incrementally updating graph triples and, therefore, of propagating changes between integrated graphs. Overall, this approach has the following benefits:

- **Formal properties**: As the algorithm fulfills all formal properties, it can not only consistently update precedence preserving graph triples of a local complete TGG, but also handle non-functional TGGs and still guarantee that correct graph triples are produced.
- **Efficiency**: Additionally, this algorithm has also a polynomial runtime complexity. This means that the algorithm performs a change propagation in relation the number of changed and affected elements and not to the number of elements in the graph in general.
- **Early cycle detection**: Compared to context-driven algorithms such as the Klar algorithm (cf. Sect. 4.6), detecting (potential) cycles along context elements can only be achieved when an actual cycle has been encountered. For example this may occur at the end of a costly transformation process. In contrast, the precedence-driven algorithm is able to determine (potential) context cycles in advance only by analyzing the precedence graph $\mathcal{PG}$.
- **Less memory consumption**: The precedence-driven batch algorithm (cf. phase (iii) of Algorithm 2) runs iteratively, and, therefore, saves memory resources. In contrast to context-driven algorithms, which use recursion, this is a clear benefit.
- **Feasibility**: Finally, it remains unclear if it is feasible to build an incremental algorithm on a context-driven basis while retaining all formal properties. With precedence-driven TGGs we definitely achieved this task.

In summary, the result of this part is that the hypothesis of this thesis (i.e, model synchronization can be achieved efficiently in combination with guaranteed formal properties) has been shown to be correct. From a formal point of view, the goals of this thesis have been fulfilled.

Discussing the drawbacks of this approach, the additional model difference recognition phase to determine Δ_S may introduce addi-

tional effort to the complete process. This is not true for the reason that in the default case it is to be expected that only a subgraph or, even better, single elements are affected by a model change. Hence, the effort of employing an additional model difference recognition phase is out ruled by the fact that only a subgraph has to be processed.

Nevertheless, the presented algorithm is by no means complete, as it is not fully implemented yet and still leaves room for further improvements. Compared to the algorithm of [KLKS10], NACs are still missing. For Algorithm 1, this feature can probably be added without problems (cf. Chap. 8). But even in the incremental case, where the synchronization process may delete elements in the output domain, it seems to be without a risk to introduce NACs to precedence TGGs: The deletion of elements can never introduce new NAC violations. As NACs demand that certain elements must not exist, deleting elements can only fulfill more NACs but not less. Thus, regarding elements that will remain as they are after a model synchronization it seems to be guaranteed that their actual transformation is still correct (but additional transformation variants could now be available).

As future work, determining affected elements more precisely compared to potentially affected elements will be one task. Recently, the algorithm processes affected elements as soon as a potential dependency has been detected. Adding and removing elements does not necessarily result in the need for untransforming other elements. This is only true for cases where elements, actually used as context, have been removed or in cases where additions result in unsatisfied dangling edge conditions. It remains to reliably identify such cases to further decrease the number of elements that have to be processed.

Furthermore, it can be argued that it would be helpful to distinguish between cases where the addition of edges together with a new node requires a retransformation of the already transformed node due to the dangling edge condition and such cases where a retransformation is only necessary to allow for the choice of now available rule application options (and could therefore be avoided). Nevertheless, the integrated graph triple in the latter case would still be consistently integrated if the retransformation is skipped. Similar arguments may hold for the deletion of an edge. Again, the result would be an improved efficiency and an extended preservation of information.

Finally, another drawback is the requirement to keep track of the applied rules in order to know which inverse rule has to be selected whenever a node has to be untransformed. It has to be researched whether it is possible to determine the inverse rule on the fly by just considering the recent elements. In cases of functional TGGs this is obviously possible. Considering that non-functionality is a desired

feature, we have to discuss other possible ways to select inverse rules appropriately. One approach could be to require that all TGG rules that are applicable to untransform a certain element delete exactly the same elements in the opposite domain. Unfortunately, this may be a too strong restriction. Another possibility could be to apply additional dangling edge checks and prohibit each inverse rule application that leads to dangling edges.

Part IV

EXTENSIONS FOR PRECEDENCE TGGS

In the following chapters, we will present three extensions that will

(i) improve the runtime efficiency and information preserving capabilities of our batch and incremental algorithms by adjusting our precedence analysis in Chap. 12 such that we *reduce the number of regarded paths* in the precedence function $\mathcal{PF}_S$.

(ii) extend expressiveness in Chap. 13 by allowing for rules with cross-domain dependencies. This will be achieved by employing a static *rule dependency analysis* which gives the control algorithms the opportunity to select an appropriate node even in cases where context dependencies only occurred in the output domain.

(iii) provide a static analysis technique in Chap. 14 to highlight potential specification problems already at compile-time. This allows for checking at compile-time if precedence cycles are possible in correctly typed graphs or not.

12

PRECEDENCE ANALYSIS WITH PATH FILTERING

In the basic precedence analysis, rules have been analyzed according to the relevant node creation patterns (cf. Def. 18). For each representation of such a pattern within a TGG rule, an appropriate entry is added to the precedence function (cf. Def. 20). This unfiltered process may induce at runtime efficiency problems and information loss and, even worse, cause unprocessable models.

The following sections present the problems with a running example and present an improved precedence analysis to overcome these.

12.1 INAPPROPRIATE DEPENDENCIES DUE TO INDIRECT PATHS

Whenever a TGG rule defines an indirect path between two types, all instances of this path have to be regarded at runtime to build the precedence graph. Regarding runtime efficiency, only shortest paths or direct connections should be regarded if available.

Example

Consider a TGG with a set of rules. In our case it is sufficient to focus on two rules, and, furthermore, analyze the source domain part only (cf. Fig. 55).

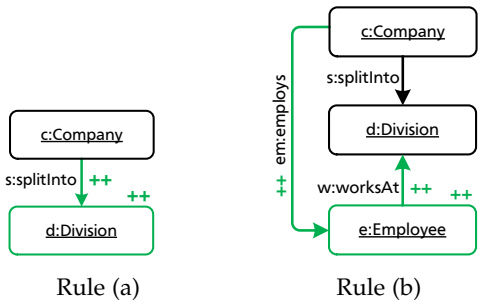


Figure 55: Source domain parts of the exemplary TGG rules that induce a direct and indirect path between `Employee` and `Division`

The following entries for the precedence function $\mathcal{PF}_S$ are retrieved according to the standard definitions of Chap. 6. For Rule (a):

- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division}) = \lessdot$

For Rule (b):

- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{worksAt}^- \cdot \texttt{Employee}) = \lessdot$

- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Employee}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{worksAt}^- \cdot \texttt{Employee}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{splitInto}^- \cdot \texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Employee}) = \lessdot$

So far, no problems arise from this situation. Unfortunately, in some cases the long paths in $\mathcal{PF}_S$ may lead to an inefficient and information threatening behavior of the incremental algorithm. Consider the source domain model depicted in Fig. 56.

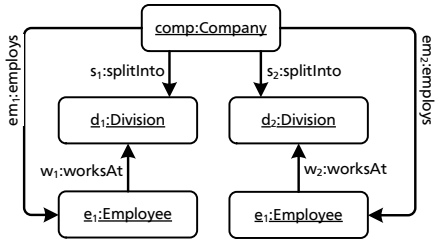


Figure 56: Source domain model

When this model is used as input for the algorithms presented in Parts ii and iii, the precedence graph $\mathcal{PG}_S$ will be computed first (depicted in Fig. 57).

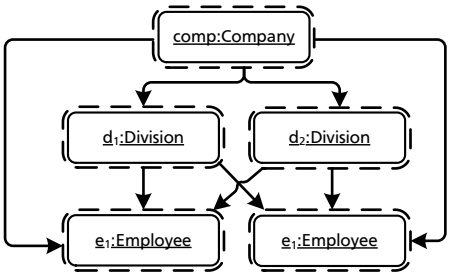


Figure 57: Precedence graph $\mathcal{PG}_S$ for the source domain model of Fig. 56

The problem that arises here is that both `Employees` $\texttt{e}_1$ and $\texttt{e}_2$ depend on both `Divisions` $\texttt{d}_1$ and $\texttt{d}_2$. This happens due to the fact that there exists a long path from $\texttt{d}_1$ via `comp` to $\texttt{e}_2$ and from $\texttt{d}_2$ via `comp` to $\texttt{e}_1$. Hence, any `Employee` can be processed right after all `Divisions` have been processed successfully but not earlier.

In the case of a batch transformation, this is not a problem as all elements will still be processed even if it is an unnecessary restriction that `Employee` $\texttt{e}_1$ can only be processed after `Division` $\texttt{d}_2$ has been processed although the transformation of $\texttt{e}_1$ will never use $\texttt{d}_2$ as a context element.

In the incremental case, the situation changes. Consider the model has been transformed appropriately and a change occurs at $\texttt{d}_2$. Algorithm 2 (cf. Chap. 10) determines all dependent elements by utilizing the precedence graph $\mathcal{PG}_S$. Hence, the algorithm will discover that `Employee` $\texttt{e}_1$ depends on $\texttt{d}_2$ and, therefore, its previous transformation has to be revoked. This is definitely an unnecessary action

which may again result in further rule revocations. As the number of dependent elements influences the efficiency of the incremental algorithm (cf. Theorem 2) the here presented situation may significantly decrease efficiency. In addition, consider non-functional TGGs, where certain heuristics or user input was utilized to decide which rule is to be used to transform the `Employee` elements. As e_1 does not really depend on d_2 it is unnecessary to revoke the transformation of e_1 and, therefore, lose the additionally introduced information.

12.2 UNPROCESSABLE MODELS DUE TO INDIRECT PATHS

Even worse, compared to efficiency and information loss issues, the same effect can lead to unprocessable models due to an apparent cycle in the precedence graph.

Example

Consider the rule depicted in Fig. 58.

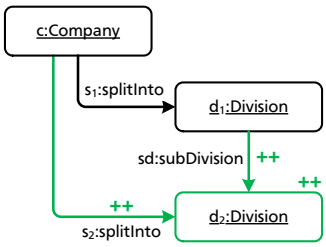


Figure 58: Problematic rule introducing sub-`divisions`

The derived entries for the precedence function $\mathcal{PF}_S$ would be as follows:

- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{subDivision}^+ \cdot \texttt{Division}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{splitInto}^- \cdot \texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{subDivision}^+ \cdot \texttt{Division}) = \lessdot$

Again, the precedence function induces no problematic or erroneous entries, but in combination with certain models, the derived precedence graph $\mathcal{PG}_S$ cannot be used to process this model as input.

Consider the model depicted in Fig. 59. Analyzing this model and deriving the precedence graph $\mathcal{PG}_S$ (depicted in Fig. 60) encounters a precedence cycle in $\mathcal{PG}_S$.

The cycle is a result of the fact that when building the precedence graph $\mathcal{PG}_S$ all paths according to the precompiled precedence function $\mathcal{PF}_S$ are searched in the input model. In this case, an instance of type

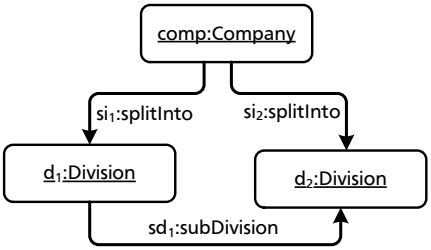


Figure 59: Source domain model

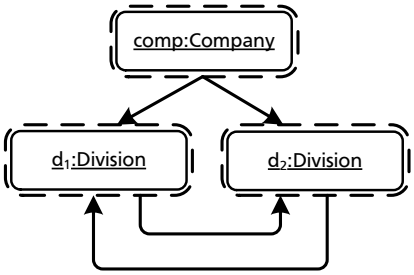


Figure 60: Precedence graph $\mathcal{PG}_S$ for the source domain model of Fig. 59

path ($\texttt{Division} \cdot \texttt{splitInto}^- \cdot \texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division}$) can be found either starting from `Division` $\texttt{d}_1$ and ending at $\texttt{d}_2$ or vice versa. Hence, two edges are added to the precedence graph $\mathcal{PG}_S$ that express both `Divisions` depend on each other, which is definitely wrong.

12.3 PATH FILTERING

In order to cope with these issues, it seems to be a suitable way to restrict the actual paths which are stored within the precedence function. The idea is that it is sufficient for each TGG rule to regard only one relevant path according to Def. 18 between two nodes even if more paths are available. If more than one path exist between the same pair of nodes, such a path selection should regard the following requirements:

- avoid precedence conflicts within the precedence function $\mathcal{PF}_S$
- avoid building cyclic precedence graphs $\mathcal{PG}_S$
- consider only such paths that are cheap to evaluate at runtime

Possible heuristics to achieve this seem to be to (i) regard shortest paths only and to (ii) prefer composition and aggregation edges over their contents.[1].

In the following, we will focus on (i) to show that such an improvement is possible. Therefore, we will restrict the consideration of paths

1 Until now, we have not dealt with typed graphs that support composition and aggregation edges. This graph property will be introduced in Chap. 14.

in a TGG rule in such a way that whenever a direct path exists (i.e., a path that traverses one edge only) between two elements that fit to one of the relevant patterns of Def. 18, all other paths in the same rule between these two elements can be neglected. If no such direct paths exist, it is still sufficient to consider only the shortest path.

Theorem 3 (Path Filtering)
Given a $\mathsf{TGG} = (\mathsf{TG}, \mathcal{R})$ *together with its set of rules* $\mathcal{R}$*. Regarding each rule* $r \in \mathcal{R}$*, it is sufficient to consider only the shortest path between each pair of nodes with respect to the relevant patterns of Def. 18.*

The computed precedence function maps only a subset of the originally mapped paths. Therefore, it has to be shown that these entries are sufficient to compute appropriate precedence relations $\lessdot_S$ and $\doteq_S$ for any given correct model G_S (i.e., $G_S \in \mathcal{L}(\mathsf{TGG})$) and, hence, derive an appropriate precedence graph $\mathcal{PG}_S$.

Proof
Considering the given $\mathsf{TGG} = (\mathsf{TG}, \mathcal{R})$ all rules have to be processed by the improved precedence analysis. Let us suppose by analyzing a rule $r \in \mathcal{R}$ a path tp is ignored in favor of tp' as this is the shortest path for a relevant pattern between the same two nodes.

At runtime, the algorithm shall transform a given graph triple, with G_S being its input. Consider rule r could be used at runtime to transform node $n' \in V_{G_S}$ where $n \in V_{G_S}$ is required as context to process n'. Thus, assume there exists a path p with $type(p) = tp$ between nodes n and n' but there exists no path p' with $type(p') = tp'$ (i.e., a precedence $n \lessdot n'$ could be determined without but not with our optimization). As path p' does not exist in G_S we know that rule r cannot be applied to transform n' as it is impossible for rule r to match at n' and its surrounding elements. This directly leads to the conclusion that either there exists another rule $r' \in \mathcal{R}$ such that node n' can be processed or G_S is not a valid input and, therefore, cannot be processed with the provided TGG. □

Example

Consider the two (partial) TGG rules of Fig. 55. Nothing changes when analyzing Rule (a), but for Rule (b) one entry will no longer be regarded: The type path $(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{worksAt}^- \cdot \texttt{Employee})$ can be neglected because a shorter path exists between a `Company` and an `Employee`: $(\texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Employee})$. The precedence function $\mathcal{PF}_S$ maps both paths onto the symbol $\lessdot$ and, therefore, it is safe to ignore the longer path. In addition, $\mathcal{PF}_S$ is no longer defined for the even more problematic path between `Division` and `Employee` via `Company`, as it is sufficient to regard the direct path $(\texttt{Division} \cdot \texttt{worksAt}^- \cdot \texttt{Employee})$.

As a consequence, the computed precedence graph $\mathcal{PG}_S$ for the source domain model depicted in Fig. 56 will no longer contain the problematic edges between d_1 and e_2 and d_1 and e_1 as paths between `Divisions` and `Employees` via a `Company` are no longer considered.

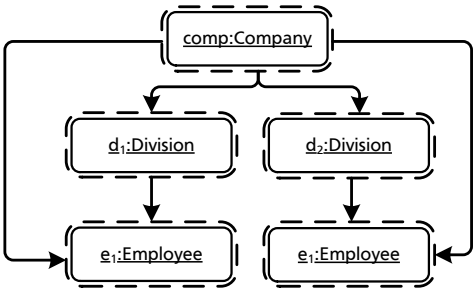


Figure 61: Reduced precedence graph for the input model of Fig. 56 with considering only shortest paths

Thus, in the case of the problematic TGG rule depicted in Fig. 58, the optimization by Theorem 3 leads to an improved precedence graph $\mathcal{PG}_S$ for the source domain model of Fig. 59. The precedence graph, depicted in Fig. 62, no longer contains a cycle between the `Divisions` but only one dependency stating that `Division` d_2 depends on d_1 as d_2 is the subdivision of d_1.

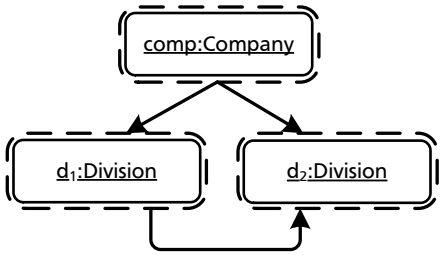


Figure 62: Reduced precedence graph for the input model of Fig. 59 with considering only shortest paths

12.4 CONGENERIC PATH FILTERING

The optimization presented above solves three problems at once: (i) efficiency is improved as less paths have to be considered at runtime without loss of generality, (ii) user added information is preserved, and (iii) the expressiveness of the presented approach is improved as certain patterns no longer result in cyclic precedence graphs.

Nevertheless, selecting the shortest path only can still result in performance and information loss issues: If paths between objects of the same equivalence class in $\mathcal{PG}_S$ are defined along objects of other equivalence classes it can happen that (real) separate equivalence classes

are merged together. Consequently, changes of (real) independent elements causes additional processing overhead as all elements of an equivalence class have to be processed in the incremental case.

To overcome this issue, we will see that it is sufficient to regard such paths that are only compounded of create objects.

Example

Consider the source domain component of a TGG rule as depicted in Fig. 63.

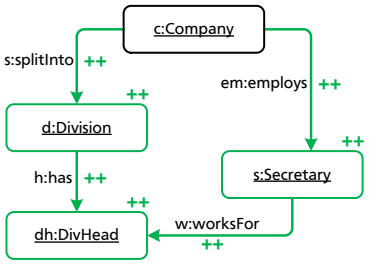


Figure 63: Source component of a TGG rule with two paths between each pair of created elements

The TGG rule states that each `Division` of a `Company` is lead by a DivisionHead (i.e., `DivHead`) who is supported by a `Secretary`. Therefore, the precedence function $\mathcal{PF}_S$ consists of the following entries:

- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{has}^+ \cdot \texttt{DivHead}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{has}^+ \cdot \texttt{DivHead} \cdot \texttt{worksFor}^- \cdot \texttt{Secretary}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Secretary}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Secretary} \cdot \texttt{worksFor}^+ \cdot \texttt{DivHead}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Secretary} \cdot \texttt{worksFor}^+ \cdot \texttt{DivHead} \cdot \texttt{has}^- \cdot \texttt{Division}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{has}^+ \cdot \texttt{DivHead}) = \doteq$
- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{splitInto}^- \cdot \texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Secretary} \cdot \texttt{worksFor}^+ \cdot \texttt{DivHead}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{has}^+ \cdot \texttt{DivHead} \cdot \texttt{worksFor}^- \cdot \texttt{Secretary}) = \doteq$
- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{splitInto}^- \cdot \texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Secretary}) = \doteq$
- $\mathcal{PF}_S(\texttt{Secretary} \cdot \texttt{worksFor}^+ \cdot \texttt{DivHead}) = \doteq$
- $\mathcal{PF}_S(\texttt{Secretary} \cdot \texttt{employs}^- \cdot \texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{has}^+ \cdot \texttt{DivHead}) = \doteq$

So far, no problems arise from this setup.

Consider an appropriate source domain model that shall be processed with a precedence-driven TGG algorithm (Fig. 64). For this reason, the precedence graph $\mathcal{PG}_S$ has to be computed and the equivalence classes are determined: $\{\texttt{comp}\}$ and $\{\texttt{d}_1, \texttt{dh}_1, \texttt{s}_1, \texttt{d}_2, \texttt{dh}_2, \texttt{s}_2\}$.

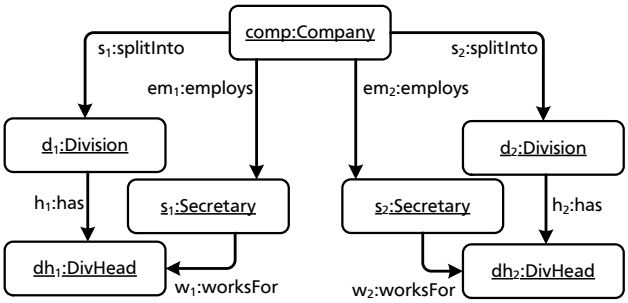


Figure 64: Source domain model with two `Divisions` (each with a `Secretary` and a `DivHead`)

The questions are (i) why are there two but not three equivalence classes and (ii) is this a problem? Unfortunately, this leads to the same efficiency and information loss problems as with long paths compared to shortest paths: Whenever an incremental change of the `Divisions`, `Secretaries`, or `DivHeads` has to be propagated, all other remaining `Divisions`, `Secretaries` and `DivHeads` have to be untransformed by applying the inverse forward rule of the TGG rule depicted in Fig. 63. Hence, additional and unnecessary computation overhead is induced, which also destroys previously introduced additional information regarding the selection of rules in cases of non-functional TGGs.

The reason why all `Divisions`, `DivHeads`, and `Secretaries` have been placed in one single equivalence class is that the create patterns between `Division` and `DivHead`, `Division` and `Secretary`, and `DivHead` and `Secretary` can be found for paths that include `Company`. In practice such a path often exists between elements that should not be processed together (e.g., $\mathtt{d}_1$ together with $\mathtt{dh}_2$).

Theorem 4 (Congeneric Path Filtering)
Given a $\mathsf{TGG} = (\mathsf{TG}, \mathcal{R})$ *together with its set of rules* $\mathcal{R}$*. For each rule* $\mathrm{r} = (\mathrm{L}, \mathrm{R})$*, it is sufficient to consider* congeneric paths *between each pair of created elements only (if such a path exists).*
A congeneric path *is a path between two nodes that lies completely in the right hand side* $\mathrm{R} \setminus \mathrm{L}$ *of* r *(i.e., consists of create nodes and edges only).*

Proof
The same arguments as for Theorem 3 can be applied here. □

Example

Reconsidering the exemplary TGG rule of Fig. 63 the following entries for $\mathcal{PF}_S$ are retrieved if congeneric paths are preferred just as stated in Theorem 4:

- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{has}^+ \cdot \texttt{DivHead}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{has}^+ \cdot \texttt{DivHead} \cdot \texttt{worksFor}^- \cdot \texttt{Secretary}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Secretary}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Secretary} \cdot \texttt{worksFor}^+ \cdot \texttt{DivHead}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{employs}^+ \cdot \texttt{Secretary} \cdot \texttt{worksFor}^+ \cdot \texttt{DivHead} \cdot \texttt{has}^- \cdot \texttt{Division}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{has}^+ \cdot \texttt{DivHead}) = \doteq$
- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{has}^+ \cdot \texttt{DivHead} \cdot \texttt{worksFor}^- \cdot \texttt{Secretary}) = \doteq$
- $\mathcal{PF}_S(\texttt{Secretary} \cdot \texttt{worksFor}^+ \cdot \texttt{DivHead}) = \doteq$

All entries via `Company` have been ignored. Hence, considering the input model of Fig. 64, the precedence graph will now consist of three equivalence classes: $\{\texttt{comp}\}$, $\{\texttt{d}_1, \texttt{dh}_1, \texttt{s}_1\}$, and $\{\texttt{d}_2, \texttt{dh}_2, \texttt{s}_2\}$.

12.5 2-PASS PATH FILTERING

The improvements presented in this chapter so far used the redundancy of paths within a single rule in order to cope with erroneous cycles in $\mathcal{PG}_S$ and, furthermore, to enhance information preserving capabilities and efficiency at runtime. The latter motivation originates from the knowledge that the costs of matching paths in models grow polynomially with the length of a path to be matched: Finding all matches for paths of length 1 (i.e., one edge has to be traversed) in a model has runtime complexity $\mathcal{O}(n)$, where n is the number of adjacent edges of the starting node. With each additional edge that has to be traversed to match a path, the runtime complexity grows: The worst-case complexity for evaluating paths of length k in a complete graph is $\mathcal{O}(n^k)$. For this reason, it is desired to traverse paths only as short as possible.

In this section, we will see that TGG rules often provide the same precedence information between two nodes that can be retrieved from the precedence graph automatically due to its transitive character. As a consequence, we will show that it is sufficient to ignore such paths in the precedence function whose information can be reconstructed from two or more shorter paths that are also regarded by the precedence function.

Example

Consider the two TGG rules as depicted in Fig. 65 (source domain parts only).

According to the definitions of the precedence analysis, the precedence function $\mathcal{PF}_S$ will contain the following entries:

- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division}) = \lessdot$

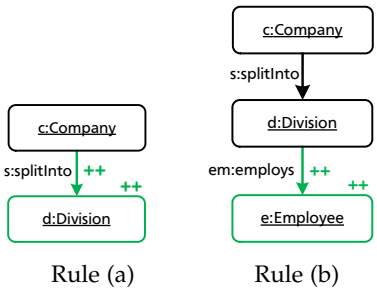


Figure 65: Source domain parts of exemplary TGG rules

- $\mathcal{PF}_S(\texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Employee}) = \lessdot$
- $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Employee}) = \lessdot$

When using this precedence function to build a precedence graph $\mathcal{PG}_S$ for a given source domain model an edge will be added between a `Company` and each of its `Employees`. Although this is not false but redundant as each `Division` depends on its `Company` and each `Employee` depends on its `Division` (i.e., appropriate edges are contained in $\mathcal{PG}_S$).

In practice, when using the precedence graph $\mathcal{PG}_S$ for determining an appropriate traversal order, such dependencies do not have to be recognized: In order to process an `Employee` its `Division` and `Company` have to be processed first. To process its `Division`, again, the `Company` has to be processed first. Hence, the dependency information is redundant and, therefore, can be safely ignored. This can be achieved via filtering redundant paths in a second iteration with an already pre-compiled precedence function $\mathcal{PF}_S$. This process is denoted as *2-pass path filtering*.

Theorem 5 (2-Pass Path Filtering)
Given a $\mathsf{TGG} = (\mathsf{TG}, \mathcal{R})$ *together with its appropriate set of rules* $\mathcal{R}$*. Regarding all entries of the same kind (i.e.,* $\lessdot$ *or* $\doteq$*) in the derived precedence function* $\mathcal{PF}_S$*, those entries can be safely removed in a second iteration that are a sequential composition of two or more other entries.*

Proof
Relations $\lessdot_S$ and $\doteq_S^*$ are transitive. Therefore, it is sufficient to consider only such paths that cannot be composed from other paths. Such composed paths do not add any information that cannot be determined by transitively computing precedences between nodes. □

Example

Considering the exemplary set of rules from Fig. 65, the last entry (i.e., $\texttt{Company} \cdot \texttt{splitInto}^+ \cdot \texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Employee}$) in $\mathcal{PF}_S$ is

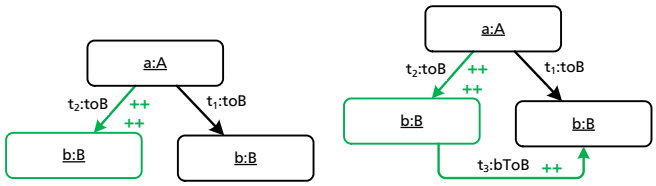
a composition of the first two entries. As described above, this entry can be ignored and, hence, no such path instance has to be searched in any concrete input model.

12.6 AUTOMORPHISMS IN TGG RULES

Altogether, these optimizations allow for a more efficient and information preserving model transformation with precedence TGGs. This should significantly improve the applicability of the approaches presented in this thesis.

Nevertheless, neither optimized precedence TGGs nor the context-driven approach is capable of handling TGG specifications that contain rules with automorphisms. This sections is intended to draw attention to this fact but will not provide a solution.

Consider a rule such as depicted in Fig. 66(a). Such a rule contains a symmetric context-pattern where the mirrored parts are not connected directly (cf. Fig. 66(a)).



(a) Problematic rule fragment (b) Unproblematic rule fragment

Figure 66: Problematic and unproblematic rule fragments for context- and precedence-driven TGG algorithms

Such a pattern states that an instance of type `B` depends on another instance of type `B`. Unfortunately, when processing a model with two instances of type `B` it is impossible to determine which element should be processed first. In the context-driven case, every instance of `B` demands that all other instances of `B` (connected to the same instance of type `A`) have to be processed first. This directly leads to a transformation cycle and the algorithm exits (cf. Sect. 4.6) with an exception. In the precedence-driven case, an instance of the type path $(\mathtt{B} \cdot \mathtt{toB}^{-} \cdot \mathtt{A} \cdot \mathtt{toB}^{+} \cdot \mathtt{B})$ can be found between any pair of instances of `B` and, therefore, a precedence cycle is introduced in $\mathcal{PG}_S$ as soon as two or more instances of `B` are connected to the same instance of `A`.

Informally, a TGG rule $r = (L, R)$ contains a problematic pattern if an automorphism $h : R \to R$ exists such that: $e \in L$ and $h(e) \in R \setminus L$.

Luckily, we know that as soon as a direct connection between b_1 and b_2 exists in the TGG rule (cf. Fig. 66(b)), the problem vanishes for

both algorithmic approaches:[2] In the context-driven case, the opposite (already processed) instance of any B is now uniquely identifiable (up to isomorphism). In the precedence-driven case, it is guaranteed in combination with the improved path considerations of Sect. 12.3 that appropriate instances of B are only found by searching for instances of path $(\mathtt{B} \cdot \mathtt{bToB}^{+} \cdot \mathtt{B})$. Therefore, precedence cycles in this context can no longer occur for any input model G_S that is a schema-compliant regarding the TGG (i.e., $G_S \in \mathrm{proj}_S(\mathcal{L}(\mathrm{TGG}))$).

Finally, it is possible to statically check whether such a problematic pattern has been specified. Hence, users that specified a TGG can be provided with sophisticated feedback about threatening TGG rules.

2 Note that if mirrored (direct) edges are introduced to the TGG rule the problem still exists.

13

RULE DEPENDENCY ANALYSIS

In this chapter, a rule dependency analysis is presented that partially solves the problem of cross-domain context dependencies caused by context elements in the domain under construction. As having such dependencies in a TGG means that this TGG is not locally complete (cf. Sect. 4.6), providing a solution to this issue improves the expressiveness.

Example

Consider the rule specifications of our running example in Chap. 3 (repeated in Fig. 67) and the input graph depicted in Fig. 68.

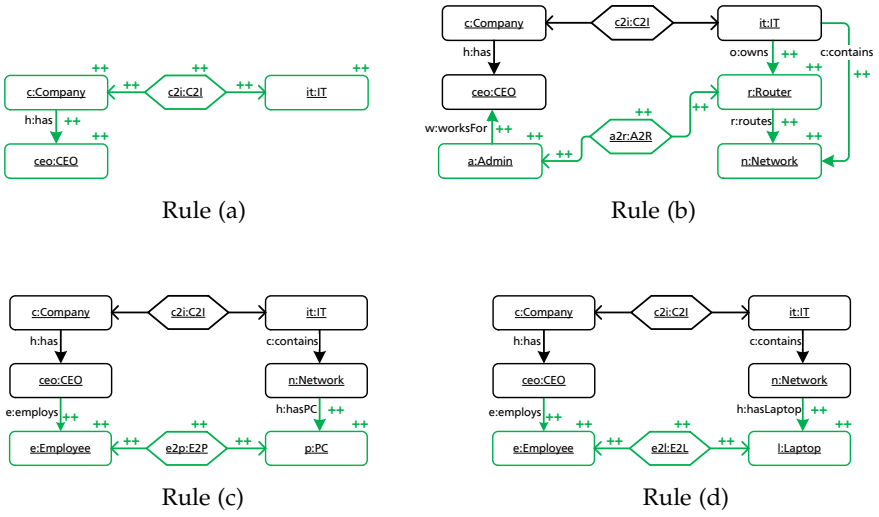


Figure 67: Set of not local complete TGG Rules for the integration

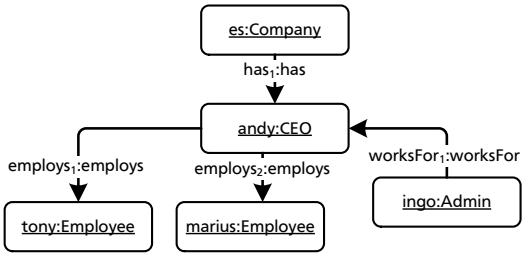


Figure 68: Input graph G_S

Algorithm 1 sorts the nodes according to the precedence function $\mathcal{PF}_S$ and determines, after transforming nodes `es` and `andy`, the set

$readyNodes = \{\texttt{ingo}, \texttt{tony}, \texttt{marius}\}$. If the algorithm randomly decides to transform node `ingo` first everything will be fine. Unfortunately, if the algorithm picks one of the nodes `tony` or `marius` first, the checks for applicability succeed (i.e., DEC holds as well as both nodes are locally transformable (cf. Sect. 4.6)), but the actual transformation will fail as no appropriate context element of type `Network` will be found in the target domain. Hence, the decision which node should be transformed next was not correct based on a local consideration only and, therefore, the local completeness criterion is violated by this TGG.

In order to cope with this issue, different approaches are possible:

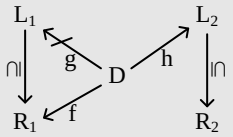
- Compute precedence functions for both source and target domains and check statically whether these functions induce contradictory precedences between related elements.
- Compute a global precedence function that considers not only source or target domains but all domains at once. Precedences that are induced just by a single domain are automatically transferred to all other domains as well.
- Besides precedences, a property named *parallel/sequential independence* [EEPT06, MKR06, MvdSD06] can be statically used to sort rules (and not elements).

To extend the class of TGGs that can be handled by Algorithm 1, an additional analysis based on the concept of *parallel/sequential independence* [EEPT06, MKR06, MvdSD06] will be introduced in this chapter and, finally, the batch algorithm will be extended appropriately.

13.1 RULE DEPENDENCY RELATION

To handle cross-domain context dependencies, we use the concept of parallel/sequential independence [EEPT06, MKR06, MvdSD06] to statically determine which rules depend on other rules. The intuition is that a rule r_2 depends on another rule r_1, if r_1 creates elements that r_2 requires as context. In other words, the post-condition (right-hand side of rule r_1) is at least partially a pre-condition of rule r_2 (left-hand side) [MKR06].

Definition 29 (Rule Dependency Relation $\lessdot_R$)
Rule $r_2 = (L_2, R_2)$, r_2 is sequentially dependent on rule $r_1 = (L_1, R_1)$ iff a graph D and morphisms f, h exist, such that there exists no morphism g as depicted below, i.e., at least one element required by r_2 (an element in L_2), is created by r_1 (this element is in R_1 but not in L_1).



The precedence relation $\lessdot_R \subseteq \mathcal{R} \times \mathcal{R}$ is defined for a given TGG as follows: $r_1 \lessdot_R r_2 \Leftrightarrow r_2$ is sequentially dependent on r_1.

In practice, $\lessdot_R$ can be statically calculated by determining all possible intersections of R_1 and L_2. If at least one element in an intersection is not in L_1 then r_2 is sequentially dependent on r_1 (i.e., $r_1 \lessdot_R r_2$).

Example

For the running example, all combinations between the rules of the TGG have to be analyzed. Starting with Rule (a) and Rule (b) (cf. Fig. 67), the intersection between the right-hand side of Rule (a) (i.e., R_a) and the left-hand side of Rule (b) (i.e., L_b) is computed and depicted in Fig. 69.

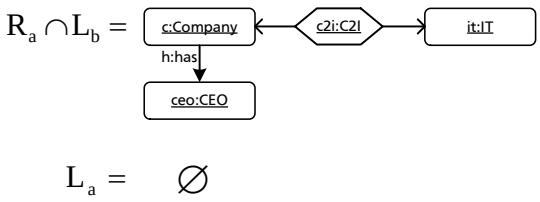


Figure 69: Intersection between R_a and L_b

According to Def. 29, two rules are sequentially dependent if the intersection has at least one element that is not in the left-hand sides of both rules. As $R_a \cap L_b \not\subseteq L_a$ it is concluded that $\mathtt{Rule}\,(a) \lessdot_R \mathtt{Rule}\,(b)$.

Next, the dependency between Rules (a) and (c) is considered. The intersection (cf. Fig. 70) is not empty, while the left-hand side L_a of Rule (a) is empty.

As $R_a \cap L_c \not\subseteq L_a$ it is concluded that $\mathtt{Rule}\,(a) \lessdot_R \mathtt{Rule}\,(c)$.

A similar result is retrieved for the dependency between Rules (a) and (d). The corresponding intersection (depicted in Fig. 71)) contains again the left-hand side L_d as a whole.

As $R_a \cap L_d \not\subseteq L_a$ it is concluded that $\mathtt{Rule}\,(a) \lessdot_R \mathtt{Rule}\,(d)$.

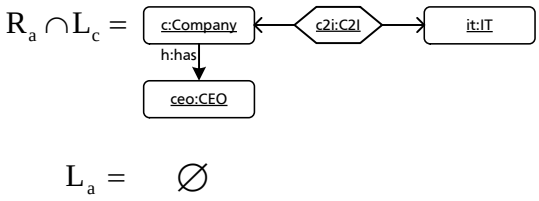


Figure 70: Intersection between R_a and L_c

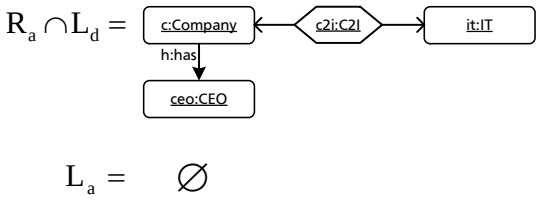


Figure 71: Intersection between R_a and L_d

Next, the dependency relationship between Rule (b) and Rule (a) is analyzed. As the intersection $R_b \cap L_a = \emptyset$, obviously no element exist in the intersection which are not in L_a. Therefore, no entry is added to relation $\prec_R$.

To determine the relationship between Rule (b) and Rule (c), the intersection of R_b and L_c is computed according to Fig. 72.

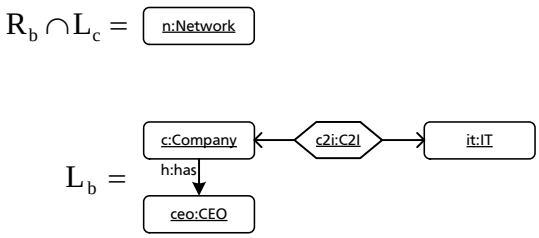


Figure 72: Intersection between R_b and L_c

As the element `n : Network` is not part of the left-hand side L_b of Rule (b), the entry $\mathrm{Rule}\,(b) \prec_R \mathrm{Rule}\,(c)$ is added.

Considering the relationship between Rules (b) and (d) a similar intersection is retrieved (cf. Fig. 73).

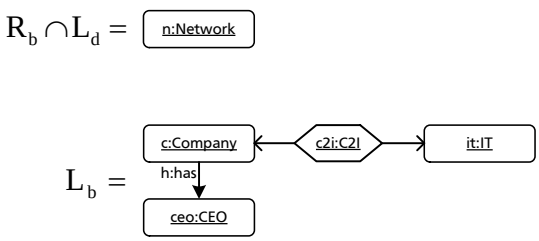


Figure 73: Intersection between R_b and L_d

Again, as L_b does not contain element `n : Network`, it is concluded that $\text{Rule}\,(b) \lessdot_R \text{Rule}\,(d)$.

For the rest of the possible entries in relation $\lessdot_R$ the following intersections have to be computed: $R_c \cap L_a$, $R_c \cap L_b$, $R_c \cap L_d$, $R_d \cap L_a$, $R_d \cap L_b$, and $R_d \cap L_c$. As all intersections are empty no additional dependencies can be determined.

Regarding additionally the potential self-dependencies, the intersections $R_a \cap L_a$, $R_b \cap L_b$, $R_c \cap L_c$, and $R_d \cap L_c$ are calculated. In all cases the intersections are equal to the left-hand side of the corresponding rule and, hence, no additional dependencies can be determined.

Altogether, for the TGG rules of this example (Fig. 67), the following pairs of rules constitute $\lessdot_R$:
$\text{Rule}\,(a) \lessdot_R \text{Rule}\,(b)$, $\text{Rule}\,(a) \lessdot_R \text{Rule}\,(c)$, $\text{Rule}\,(a) \lessdot_R \text{Rule}\,(d)$, $\text{Rule}\,(b) \lessdot_R \text{Rule}\,(c)$, and $\text{Rule}\,(b) \lessdot_R \text{Rule}\,(d)$.

13.2 THE EXTENDED CONTROL ALGORITHM

Algorithm 1 (cf. Chap. 7) will be extended with an additional state for determining the appropriate node transformation order based upon relation $\lessdot_R$. Therefore, all nodes that are ready for transformation are sorted according to the relation $\lessdot_R$. This additional step either optimizes the partial order such that all nodes can be transformed appropriately or does no harm, meaning that in case where no optimization can be applied the algorithm acts as if the relation $\lessdot_R$ was not utilized.[1]

Algorithm 1 of Chap. 7 has been extended with a command on line (5). Here, the set *readyNodes* is sorted according to the partially ordered relation $\lessdot_R$, i.e., the rules that can be used to transform nodes in *readyNodes* are determined, sorted according to $\lessdot_R$ and reflected in *readyNodes*. This could be achieved by assigning an integer to each rule according to the partial order of $\lessdot_R$ and then selecting the largest number of all rules that translate $n \in$ *readyNodes* for n. If it is not possible to sort *readyNodes* due to cycles in $\lessdot_R$, this additional analysis supplies no further information and *readyNodes* remains unchanged. The rest of the algorithm remains as presented in Chap. 7.

Example

To demonstrate the extended algorithm, we apply the same forward

1 In cases of cycles an appropriate heuristics could be to break the cycle arbitrarily and try to transform the graph. As the cycle already induces problems with the TGG (i.e., all elements in `readyNodes` should be processable regardless of their relative order), it cannot be guaranteed that the transformation succeeds with or without this sorting.

Algorithm 3 Precedence TGG Batch Algorithm

```
1: procedure TRANSFORM(G_S, ≺_R, PF_S)
2:   PG_S ← BUILDPRECEDENCEGRAPH(G_S, PF_S)
3:   while (PG_S contains equivalence classes) do
4:     readyNodes ← all nodes in equiv. classes in PG_S
                    without incoming edges
5:     readyNodes ← sort readyNodes using ≺_R
6:     select first node n from readyNodes
7:     transformedNodes ← CHOOSEANDAPPLYRULE(n)
8:     if transformedNodes ≠ ∅ then
9:       PG_S ← remove all nodes in transformedNodes from PG_S
10:    else if transformedNodes = ∅ then
11:      terminate with exception
12:    end if
13:  end while
14:  return G_S ← G_C → G_T
15: end procedure
```

transformation for the source graph of our example triple depicted in Fig. 33. Given as input the source graph G_S, the rule dependency relation $\prec_R$ (depicted as a graph in Fig. 74(b)), and the precedence function $\mathcal{PF}_S$ (cf. example for Def. 20). On line (2), the precedence graph $\mathcal{PG}_S$ for G_S, depicted in Fig. 74(a), is built. $\mathcal{PG}_S$ is acyclic, hence the transformation can continue.

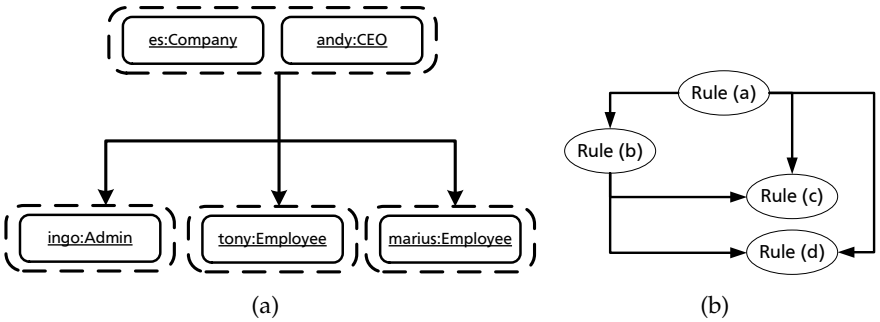


Figure 74: $\mathcal{PG}_S$ for the input graph (left) and relation $\prec_R$ for all rules (a)–(d) (right)

On line (4), the set *readyNodes* is calculated, consisting in this case of the nodes `as` and `andy` from a single equivalence class of $\mathcal{PG}_S$. On line (5), only Rule (a) can be used to transform both nodes and, therefore, the sorting is trivial. On line (6) `es` or `andy` is chosen randomly, and in either case, the only candidate rule is Rule (a) (Fig. 21), which can be directly applied on line (7). Again in either case, *transformedNodes* contains both nodes as Rule (a) transforms `es` and `andy` simultaneously. $\mathcal{PG}_S$ is updated on line (9) to consist of three unconnected equivalence

classes `ingo`, `tony`, and `marius`. In the second iteration through the while-loop, *readyNodes* contains all these three elements and will be sorted according to $\prec_R$ on line (5). This time, the sorting reveals that `ingo` must be transformed before `tony` and `marius` as Rules (c) and (d) both require a `Network` as context in the target domain, which can only be created by applying Rule (b) first, i.e., $\texttt{Rule (b)} \prec_R \texttt{Rule (c)}$, $\texttt{Rule (b)} \prec_R \texttt{Rule (d)}$ (Fig. 74(b)). The algorithm select the first element in `readyNodes` on line (6), which is `ingo`. Applying Rule (b) (line (7)) puts `ingo` in *transformedNodes*, $\mathcal{PG}_S$ is updated on line (9) to now contain only `tony` and `marius`. In the third iteration, *readyNodes* contains `tony` and `marius`, and no sorting is needed as Rules (c) and (d) do not depend on each other. On line (6) `tony` could be randomly selected first and (arbitrarily or via user input) Rule (c) could be chosen to be applied on line (7). After updating $\mathcal{PG}_S$ only `marius` remains untransformed. Similar to the penultimate iteration, Rule (d) could be selected and applied this time. Updating $\mathcal{PG}_S$ on line (9) empties the precedence graph, which terminates the while-loop on line (3). The created graph triple depicted in Fig. 33 is returned on line (14).

13.3 FORMAL PROPERTIES

In this section, we prove that our algorithm retains all formal properties introduced by [Sch95, SK08] (cf. Sect. 4.3).

Theorem 6 *Algorithm 3 is correct, complete and efficient for any local complete TGG.*

Proof

Correctness:

Regarding the proof of correctness, the same arguments hold for the extended algorithm as for Algorithm 1. □

Completeness:

In addition to the proof for completeness for Algorithm 1, further arguments have to be incorporated:

Consider the algorithm with the additional relation $\prec_R$ and, therefore, the capability of handling specifications with cross-domain context dependencies as in our running example. We have shown in Sect. 4 that the algorithm presented in [KLKS10] cannot cope with such specifications as they violate the local completeness criterion. We can, hence, conclude that Algorithm 3 is more expressive than the previous context-driven algorithm as it can handle certain TGGs that are not local complete. We leave the precise categorization of this new class of TGGs to future work. □

Efficiency:
For efficiency considerations the additional line (5) has to be taken into account. Sorting a finite set of nodes can be achieved with algorithms such as merge sort or heap sort in $\mathcal{O}(n \log(n))$, where n is the number of elements in the set. In the worst-case, this set (i.e., `readyNodes`) contains all nodes of the graph. We assume that retrieving the rules that may transform a certain node is constant (i.e., $\mathcal{O}(1)$). Hence, sorting `readyNodes` has an $\mathcal{O}(n \log(n))$ complexity. As this additional effort clearly does not extend the complexity $O(n^k)$, the overall complexity remains the same but of course an additional overhead has been introduced. □

14

EXTENDED PRECEDENCE ANALYSIS

In this chapter, a third extension of precedence TGGs will be presented that employs an additional static analysis. As we have seen in Parts ii and iii, precedence cycles (i.e., a cyclic precedence graph) in models which are passed to the algorithm as input cannot be handled. Hence, the algorithms exit with an appropriate exception before actually starting the transformation process. From a user point of view, it seems to be a natural requirement to know if a TGG specification is potentially unsafe. Unsafe in this regard stands for the potential to actually process models with precedence cycles that cannot be handled by the algorithms.

As a consequence, the contribution of this chapter is a static analysis on the type level, which analyzes a given TGG at compile-time and determines if precedence cycles may occur in schema-compliant graphs at runtime. We are focused only on schema-compliant graphs, and not on the larger set of correctly typed graphs and, therefore, provide a reliable means for potential threats.

Furthermore, this approach is based upon additional categories of edge types: inheritance, composition, and aggregation edges. Hence, this chapter will bring the basic formulation of type graphs of Chap. 3 to a more expressive level, where abstract node types, inheritance between node types, and containment relations are available.

14.1 EXTENDED TYPE GRAPHS AND TYPED GRAPHS

In order to apply the desired analysis, the concept of type and typed graphs has to be extended with additional features. Type graphs with inheritance and graph morphisms are defined according to the common notation presented in [EEPT06]. In addition, the work of Biermann et al. [BET08] served as an inspiration for the definition of graphs with composition and aggregation relationships.

According to standard UML semantics [OMG11], the following definition introduces abstract types, inheritance between types, aggregation and composition.

Definition 30 (Extended Type Graphs)
$ET = (TG, A_{V_{TG}}, N_{E_{TG}}, I_{E_{TG}}, C_{E_{TG}}, A_{E_{TG}})$ *is an* extended *type graph consisting of a standard type graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ *(cf. Def. 2) such that*

- V_{TG} *denotes a set of node types*
- E_{TG} *a set of edge types and* s_{TG}, t_{TG} *the source and target functions*
- *a set* $A_{V_{TG}} \subseteq V_{TG}$ *of abstract node types*
- *a set* $N_{E_{TG}} \subseteq E_{TG}$ *of normal edge types*
- *a set* $I_{E_{TG}} \subseteq E_{TG}$ *of inheritance edge types, where* $(\forall i \in I_{E_{TG}} : s(i)$ *is subtype* $\wedge\, t(i)$ *is supertype)*
- *a set* $C_{E_{TG}} \subseteq E_{TG}$ *of composition edge types, where* $(\forall c \in C_{E_{TG}} :$ $s(c)$ *is container* $\wedge\, t(c)$ *is content)*
- *a set* $A_{E_{TG}} \subseteq E_{TG}$ *of aggregation*[a] *edge types, where* $(\forall a \in A_{E_{TG}} :$ $s(a)$ *is container* $\wedge\, t(a)$ *is content)*

Since edge types can be of four different kinds we define $N_{E_{TG}}, I_{E_{TG}}, C_{E_{TG}}, A_{E_{TG}}$ *as pairwise disjunct and* $N_{E_{TG}} = E_{TG} \setminus \{I_{E_{TG}} \bigcup C_{E_{TG}} \bigcup A_{E_{TG}}\}$.
For each node type $nt \in V_{TG}$ *the* inheritance clan *is defined by* $clan_I(nt) = \{nt_1 \in V_{TG} \mid \exists$ *sequence* $nt_1, nt_2, \ldots, nt_k = nt \in V_{TG} \wedge 1 \leqslant i < k : \exists\, et \in I_{E_{TG}}$ *such that* $s(et) = nt_i \wedge t(et) = nt_{i+1}\} \bigcup \{nt\} \subseteq V_{TG}$. *The inheritance clan of a node type* nt *denotes a set of node types that are (transitively) all subtypes of* nt *including the type itself, i.e.,* $nt \in clan_I(nt)$.

a We distinguish between composition and aggregation in order to allow for more sophisticated differentiation between two popular kinds of part-whole relationships.

Example

Considering our running example, we extend the metamodel (i.e., type graph) of the source domain with additional semantics. The extended type graph is depicted in Fig. 75: The edge between `Company` and `Division` has been adjusted from a standard edge type to *composition* (filled diamond end). This states according to standard UML semantics [OMG11] that a `Division` may never have two or more `Company` elements (single parent element), a `Civision` may never contain itself (directly or transitively), and the deletion of the parental element also results in the deletion of all children elements. In addition, the edge between `Company` and `CEO` has been changed to *aggregation* (empty diamond end) in order to express that a CEO is contained in a `Company` but `CEOs` do not stop existing as soon as a `Company` vanishes.

Additionally, a `Division` employs `Employees`, which are abstract (italic type name) meaning that in a concrete instance each `Employee` is either a `Consultant` or an `Admin`. Hence, both types `Consultant` and `Admin` inherit from `Employee`. Finally, `Divisions` may be organized in

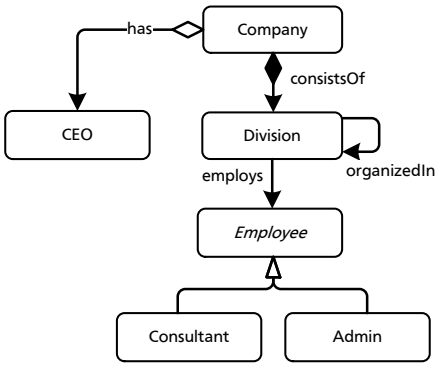


Figure 75: Extended type graph of the source domain

an arbitrary number of sub-`Divisions` as induced by the reflexive edge at `Division`.

Next, we define typed graphs with containment, i.e., graphs that comply to an extended type graph. Since we differentiate in extended type graphs between different edge kinds, typed graphs have to be defined with composition and aggregation relations as well.

Definition 31 (Typed Graph with Containment)
Given an extended type graph ET *(cf. Def. 30) and a graph* G. G *is typed over* ET *via a clan morphism [EEPT06] type* $: G \to ET$ *iff* $\forall e \in E : type_E(e) \notin I_{E_{TG}} \wedge type_V(s(e)) \in clan_I(s_T(type_E(e))) \wedge type_V(t(e)) \in clan_I(t_T(type_E(e)))$.

Containment can be realized via composition or aggregation. Therefore, a graph with containment owns a distinguished set of composition edges $C = \{e \in E \mid type_E(e) \in C_{E_{TG}}\} \subseteq E$ *and a distinguished set of aggregation edges* $A = \{e \in E \mid type_E(e) \in A_{E_{TG}}\} \subseteq E$. *Containment edges (i.e., composition and aggregation edges) induce the binary relation* directly-contained*:* $\texttt{directly-contained} = \{(x, y) \in V \times V \mid \exists\, e \in A \bigcup C : (s(e) = x \wedge t(e) = y)\}$.
Thus, relation contains *denotes the transitive closure of relation* `directly-contained`. *Furthermore, all containment edges must fulfill the following property:*[1]

(1) Containment cycles are prohibited. Formally, $\forall x \in V : (x, x) \notin$ *contains.*

Additionally, composition edges must fulfill the following property:

(2) Each content can have at most one container. Formally, $e_1, e_2 \in C : t(e_1) = t(e_2) \Rightarrow e_1 = e_2$.

Furthermore, we require that:

(3) Abstract types cannot be instantiated. Formally, $\forall n \in V : type_V(n) \notin A_{V_{TG}}$.

(4) *Inheritance edges cannot be instantiated. Formally,* $\forall e \in E : type_E(e) \notin I_{E_{TG}}$ *(no inheritance edges).*

Example

Extending our running example, the following five TGG rules have to be considered.

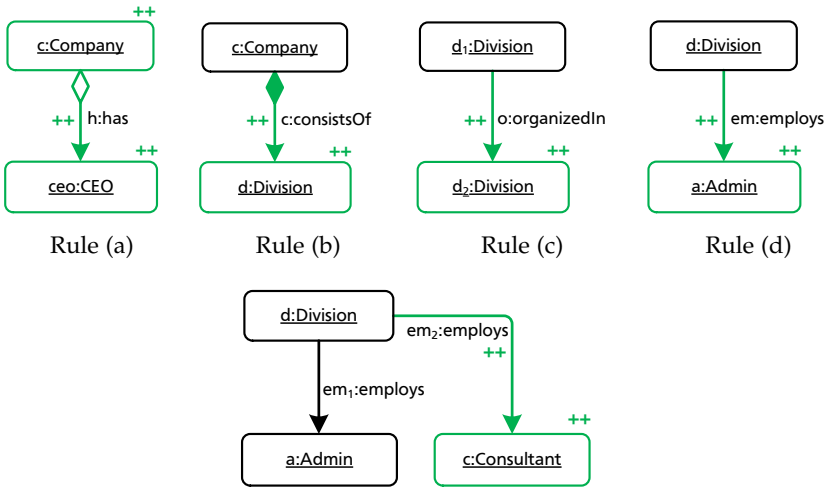


Figure 76: Extended set of TGG rules (source domain only)

Figure 76 depicts the source domain component of the five rules. Deriving the precedence function from these rules for the source domain, the following entries comprise $\mathcal{PF}_S$:
From Rule (a): $\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}$ and $\texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}$.
From Rule (b): $\texttt{Company} \cdot \texttt{consistsOf}^+ \cdot \texttt{Division}$
From Rule (c): $\texttt{Division} \cdot \texttt{organizedIn}^+ \cdot \texttt{Division}$
From Rule (d): $\texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Admin}$
From Rule (e): $\texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Consultant}$ and $\texttt{Admin} \cdot \texttt{employs}^- \cdot \texttt{Division} \cdot \texttt{employs}^+ \cdot \texttt{Consultant}$

Note that using inheritance in type graphs introduces new constraints on rules and their applications. From now on rules are also applicable to matches where instances of subclasses of actually referred elements are found. This means that context elements of a rule (i.e., nodes in L) are identifiable with instances of subtypes when applying a rule, while create elements (i.e., nodes in $R \setminus L$) have to be of exactly this type. The latter is especially important for derived operational rules (cf. Def. 13) where create elements are now treated as context elements but have to be exactly of the specified type and not of a subtype. Otherwise, the control algorithm may accept a larger

class of models that are correctly typed, but actually not producible with the set of rules.

As a consequence, derived precedence information in the precedence function $\mathcal{PF}_S$ needs to be processed for subclasses as well. This can be achieved by deriving all combinations of type paths explicitly and adding them to the precedence function $\mathcal{PF}_S$. Thus, all definitions from Chap. 6 do not have to be changed. Only the process of building the precedence function $\mathcal{PF}_S$ has to take the additional inheritance information into account.

Consider for example a TGG rule that expresses a context relationship between types A and B via edge toB pointing from A to B (i.e., an instance of A is required to process an instance of B). Hence, the precedence function $\mathcal{PF}_S$ contains an entry $\mathcal{PF}_S(A \cdot toB^+ \cdot B) = \lessdot$. Furthermore, applying this rule may also accept instances of all subclasses of A, which is reflected by additional entries in $\mathcal{PF}_S$ such that $\mathcal{PF}_S(A' \cdot toB^+ \cdot B) = \lessdot$. Also in cases the metamodel specified also subclasses for B, the TGG rule would always only create an instance of B. Consequently, derived operational rules must only process instances of B but not instances of any subclass of B.

14.2 PRECEDENCE RELATIONS ON THE TYPE LEVEL

Analogously to relation $\lessdot_S$ for a typed graph, we define two other relations $\lessdot^c_{ET_S}$ and $\lessdot_{ET_S}$, which use a previously computed precedence function $\mathcal{PF}_S$ for a given TGG to determine the precedences on the type level.

The first relation is used to collect precedence information only for containment structures, while the second relation is used to collect all precedence information. Both relations are necessary to differentiate in a further step between cycles that are caused only by containment edges (safe) and cycles that are caused by arbitrary edges (unsafe).

Definition 32 (Relation $\lessdot^c_{ET_S}$ on homogenous contains paths)
Given $\mathcal{PF}_S$, the precedence function for a given TGG*, and an extended source domain type graph ET_S. The precedence relation $\lessdot^c_{ET_S} \subseteq V_{ET_S} \times V_{ET_S}$ for the source domain is defined as follows: $n \lessdot^c_{ET_S} n'$ if there exists a type path $tp \in TPaths_S$ (cf. Def. 19) between nodes n and n' such that $\mathcal{PF}_S(tp) = \lessdot \wedge \forall e \in tp : e \in C_{E_{ET_S}} \cup A_{E_{ET_S}}$. Such a path is called* homogenous *as it traverses containment edges only.*

Definition 33 (Relation $\lessdot_{ET_S}$ on mixed paths)
Given $\mathcal{PF}_S$, the precedence function for a given TGG*, and an extended source domain type graph ET_S. The precedence relation $\lessdot_{ET_S} \subseteq V_{ET_S} \times V_{ET_S}$ for the source domain is defined as follows: $n \lessdot_{ET_S} n'$ if there exists a type path $tp \in TPaths_S$ (cf. Def. 19) between nodes n and n' such that $\mathcal{PF}_S(tp) = \lessdot$ and it exists no entry $n \lessdot^c_{ET_S} n'$. Such a path is called* mixed *as it traverses edges regardless of their types (i.e., either composition, aggregation or normal).*

These relations subsume all inherently existing precedence information from the set of rules on the type level.

Example

Regarding our running example (cf. type graph of Fig. 75), relation $\lessdot^c_{ET_S}$ consists of the tuple `Company` $\lessdot^c_{ET_S}$ `Division`.
Thus, relation $\lessdot_{ET_S}$ consists of the following tuples

- `Division` $\lessdot_{ET_S}$ `Division`
- `Division` $\lessdot_{ET_S}$ `Admin`
- `Division` $\lessdot_{ET_S}$ `Consultant`
- `Admin` $\lessdot_{ET_S}$ `Consultant`

Furthermore, we define an equivalence relation to partition the node types for the analysis.

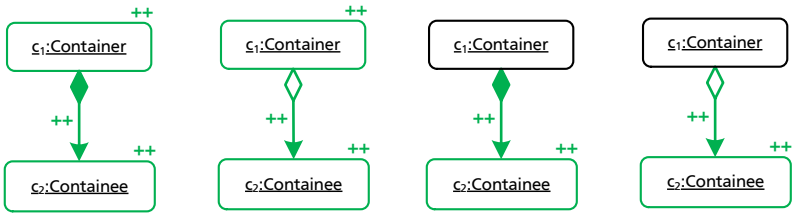
Definition 34 (Relation $\doteq_{ET_S}$)
Given $\mathcal{PF}_S$, the precedence function for a given TGG*, and an extended source domain type source graph ET_S. The symmetric relation $\doteq_{ET_S} \subseteq V_{ET_S} \times V_{ET_S}$ for the source domain is defined as follows: $n \doteq_{ET_S} n'$ if there exists a type path $tp \in TPaths_S$ between nodes n and n' such that $\mathcal{PF}_S(tp) = \doteq$.*

Definition 35 (Equivalence Relation $\doteq^*_{ET_S}$)
The equivalence relation $\doteq^*_{ET_S}$ is the transitive and reflexive closure of the symmetric relation $\doteq_{ET_S}$.

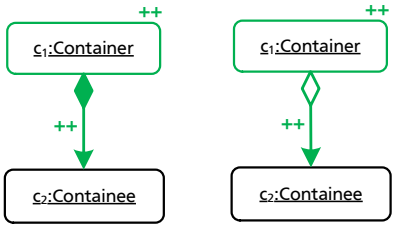
Example

For our example (Fig. 75), the following equivalence classes constitute $\doteq^*_{ET_S}$: {`Company`, `CEO`}, {`Division`}, {`Admin`}, {`Consultant`}.

Restriction: Containment hierarchies can only be built in a top-down manner (cf. Fig. 77) without reducing expressiveness. Top-down in this regard means that either containers have to be already transformed and, therefore, are used as context to append contents, or are created together by the same rule.



(a) Allowed TGG rule fragments (source or target domain)



(b) Disallowed TGG rule fragments (source or target domain)

Figure 77: Allowed and disallowed rule fragments for building a containment hierarchy

This is necessary as containment edges should be reliably ignored for statically analyzing TGG specification.

Definition 36 (Type Precedence Graph $\mathcal{PG}_{ET_S}$)
The type precedence graph $\mathcal{PG}_{ET_S}$ for a given source type graph ET_S is a graph constructed as follows:
(i) The equivalence relation $\doteq^*_{ET_S}$ is used to partition V_{ET_S} into equivalence classes $EQ_1, \ldots, EQ_n$ which serve as the nodes of $\mathcal{PG}_S$, i.e., $V_{\mathcal{PG}_{ET_S}} := \{EQ_1, \ldots, EQ_n\}$.
(ii) The edges in $\mathcal{PG}_S$ are defined as follows: $E_{\mathcal{PG}_{ET_S}} := \{e \mid s(e) = EQ_i,\ t(e) = EQ_j : \exists\, n_i \in EQ_i, n_j \in EQ_j$ with $(n_i \lessdot^c_{ET_S} n_j) \vee (n_i \lessdot_{ET_S} n_j)\}$.

Remark: $\mathcal{PG}_{ET_S}$ defines a partial order over equivalence classes. This is a direct consequence of Def. 36.

Example

Consider the type graph ET_S, depicted in Fig. 75. The type precedence graph is depicted in Fig. 78 with equivalence classes in dashed lines.

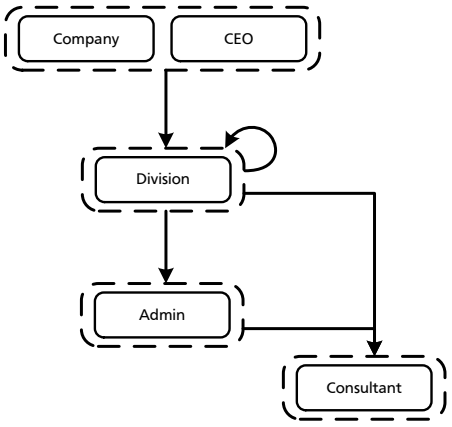


Figure 78: Type precedence graph $\mathcal{PG}_{ET_S}$

14.3 ANALYZING THE TGG SPECIFICATION

In order to use relations $\lessdot_{ET_S}$ and $\lessdot^{c}_{ET_S}$ for static analysis purposes, we show that there is a special connection between the precedence graphs on the type and on the instance level. The idea is to show that if the precedences between types have only containment cycles (i.e., $\mathcal{PG}_{ET_S}$ is acyclic for all mixed paths), we know that all instance graphs will never have any visiting order cycles and, hence, any precedence graph $\mathcal{PG}_S$ will be acyclic.

The motivation behind this is that whenever cyclic precedences on a type level exist, it cannot be guaranteed that actual model transformation processes may never be blocked due to cyclic dependencies.[2] For this reason, this static analysis helps users to specify only those TGGs that do not have such cycles at all and, hence, an actual model transformation of a correct will never be blocked due to cyclic dependencies.

For a given precedence function $\mathcal{PF}_S$ without specification errors (i.e., no entry is mapped to $\cdot\lightning\cdot$) and type graph ET_S the following theorem states the desired aspect.

Theorem 7
The precedence graph $\mathcal{PG}_S$ is acyclic for any graph G_S typed over ET_S, if the precedence graph on the type level $\mathcal{PG}_{ET_S}$ is acyclic for mixed paths of normal and containment edges.

Proof
Consider a graph G_S and a type graph ET_S, where G_S is typed over ET_S (i.e., $G_S \to ET_S$). Informally, we show that when $\mathcal{PG}_S$ is cyclic

2 This is true for context-driven transformation concepts as well as for the here presented precedence-driven transformation concept.

then $\mathcal{PG}_{ET_S}$ is also cyclic. By proving this, we indirectly show that a precedence graph with containment cycles on the type level only guarantees that all precedence graphs on the instance level are acyclic.

Furthermore, consider we computed a cyclic precedence graph $\mathcal{PG}_S$ for graph G_S. $\mathcal{PG}_S$ has a cycle of arbitrary length starting and ending at the equivalence class of node n. As $\mathcal{PG}_S$ contains only such edges that denote a path p in G_S whose type path tp exists in $TPaths_S$ (cf. Def. 25), we conclude that the type of every edge in the denoted path in $\mathcal{PG}_S$ is also part of a path on type level in $\mathcal{PG}_{ET_S}$.

$\mathcal{PG}_{ET_S}$ either consists of edges denoting homogeneous containment paths (i.e., homogeneous aggregation, composition or mixed aggregation and composition paths) or of edges denoting mixed paths. As a consequence, it is obvious that cycles along mixed edges in $\mathcal{PG}_{ET_S}$ reflect only such cycles that may really occur in graphs. This is true for two reasons: (1) Aggregation and composition cycles are prohibited on the instance level (cf. Def. 31) and (2) by restricting rules to build containment hierarchies in a top-down manner only, it is assured that every containment cycle complies to a precedence cycle.
Consequently, the assumed precedence cycle in $\mathcal{PG}_S$ consists of at least one edge e of edge type $et \in N_{E_{TG}}$, which implies that the precedence graph on the type level $\mathcal{PG}_{ET_S}$ also contains a mixed cyclic path. Otherwise, G_S is not correctly typed over ET_S (cf. Def. 31).
This directly leads to the fact that a sequence in $\mathcal{PG}_{ET_S}$ from and to nt exists with $nt \in clan_I(type_V(n))$
And the conclusion is that if no such sequence exists in $\mathcal{PG}_{ET_S}$ it is guaranteed that no cyclic sequence in $\mathcal{PG}_S$ exists either. □

Example

Regarding the running example, a self-reflexive cycle exists at the type `Division`. This is because in Rule (c) (cf. Fig. 76) a `Division` is attached to an already existing `Division`. In practice, the type graph allows for cyclic organization of `Divisions` and, hence, accepts such cycles as input models. Unfortunately, the precedence-driven algorithms (cf. Chap. 7 and 10) reject such an input model as its precedence graph $\mathcal{PG}_S$ would be cyclic. But context-driven algorithms such as [KLKS10] (cf. Sect. 4.6) can neither cope with such a cyclic dependency: context elements have to be transformed first and, therefore, the algorithm terminates with an exception. The solution would be either to eliminate Rule (c) and, therefore, reduce the size of the processable models (decrease the expressiveness) or to adjust the type of the edge from and to type `Division` to aggregation or composition.[3]
Having done this, the type precedence graph $\mathcal{PG}_{ET_S}$ has only a cycle

3 Of course, users may also ignore such a warning if they are sure that no cyclic model will ever be passed to the algorithm.

of homogenous containment edges, and, therefore, guarantees that no cycle may exist on the instance level.

14.4 COMPARISON WITH CRITICAL PAIR ANALYSIS

Static analysis and quality assurance of integration specifications are still an open issue. Recently, static analysis for TGG specifications is considered in [EEHP09, HEGO10, EHGB12]. Static analysis based upon precedence relations, as presented here, derives the needed dependency relationships between node types based on the inspection of occurrences of the regarded types in all TGG rules.

Critical pair analysis (CPA), on the other hand, is applied to specifications and computes dependency relationships between pairs of rules based on the occurrences of all types of graph elements in the regarded rules [EGLT11] and, therefore, accordingly may result in a translation dead-end. Transformation algorithms such as [BTSoo] use CPA to postpone critical rule applications in order to reduce the depth of potentially necessary backtracking. This idea significantly improves the efficiency of translation algorithms based on backtracking.

Our work is complementary to this approach since CPA does not consider ways to compute a reasonable translation order with an appropriate starting point, while static analyses based on precedence relations do not regard potential rule application errors. Thus, the here presented static analysis and CPA complement each other and may be used for rather different purposes: static analysis based on precedence relations allows one to reduce the set of processable nodes, whereas CPA allows one to reduce the set of potentially applicable rules. For this reason, static analysis based on precedence relations can be used to efficiently identify such a translation order and, thus, it improves efficiency even more.

In addition, critical pair analysis reveals pairs of rules that may translate a node with (at least) two rules. Our TGG approach overcomes this problem as long as the competing rules are applied to the same elements [KLKS10] (cf. Sect. 4.6), which means that the presented TGG approach allows for non-functional rule specifications, which cannot be handled by CPA.

15

EVALUATION OF THE EXTENSIONS FOR PRECEDENCE TGGS

In this part, three extensions for precedence TGGs have been presented. With a sophisticated path filtering approach (cf. Chap 12), a rule dependency analysis (cf. Chap. 13), and a precedence analysis on the type level (cf. Chap. 14) we achieved the following:

- **Improved efficiency and information preservation**: By considering a reduced number of paths at runtime, we improved the efficiency and information preserving capabilities of the batch and incremental algorithm.
- **Improved expressiveness (i)**: In addition, considering a reduced number of paths provably enhances the expressiveness of precedence TGGs.
- **Improved expressiveness (ii)**: Thus, expressiveness could be extended even further by employing an additional relation on rules that allows for selecting an appropriate element in cases where more than one element is available.
- **Improved static analysis**: Finally, with an extended precedence analysis on the type level, we presented a way to test TGG specifications for potential precedence cycles.

An interesting question for future work is, how path filtering can be used to avoid cycles on a (type) graph level or conflicts between path mappings in the precedence functions. Therefore, we have to investigate if path filtering should prefer longer paths instead of shorter paths in some situations to avoid precedence conflicts and/or cycles, without compromising the guaranteed formal properties of the precedence-driven algorithms.

Regarding the rule dependency analysis, we have to admit that this approach cannot solve all local completeness problems. As soon as attributes and other constraints come into play, which introduce additional constraints to the matching process, new sources for local completeness conflicts arise. Since checking for DEC (cf. Sect. 4.6) as well as our relation $\lessdot_R$ relies on type information only, conflicts between attribute values cannot be considered. From a practical point of view, it would probably too much effort to implement both, rule dependency and precedence analyses. Therefore, presenting the rule dependency analysis was intended to show that other techniques also exist that cope with dependencies.

Part V

IMPLEMENTATION

The following chapters present implementation details regarding different aspects of a TGG implementation. In Chap. 16, we start with a brief introduction to our tool eMoflon and the graph transformation approach of Story Driven Modeling (SDM) as this is used to specify the TGG transformation algorithm as well as the TGG rule derivation process.

In a second step (cf. Chap. 17), we discuss how the context-driven TGG control algorithm of [KLKS10] has been implemented and how it performs. Finally, we discuss in Chap. 18 how the TGG rule to operational rule derivation process according to Def. 13 has been realized. Both components have been implemented by using SDM (and a small portion of hand-written code).

16

INTRODUCTION TO eMoflon

The whole implementation has been realized with the meta-CASE tool eMoflon.[1] eMoflon has been developed by the Real-Time Systems Lab at the Technische Universität Darmstadt since the beginning of 2011 [ALPS11, ALS12]. The tool is conceptually based upon its predecessor MOFLON [AKRS06, KKS07, AKK+08, KRS09] but uses a different technological foundation:

- Instead of MOF a subset named Ecore is supported only. This allowed for an efficient integration into the de-facto standard development environment Eclipse.
- Furthermore, as a user interface, an addin for the Enterprise Architect (EA) offered by Sparx Systems has been programmed, which enables users to comfortably define metamodels, graph transformations and TGGs in a well-tested and mature modeling tool.

The overall architecture and design decisions have been discussed in [ALPS11]. Fig. 79 depicts the basic components of eMoflon and their relationships.

The upper part denotes the graphical user interface provided by the EA addin. The lower left part shows how metamodels and other specifications are processed with different code generators and compilers. Finally, the lower right part mentions some components provided on top of the compiled Java code. Such components include a supporting framework for user assistance (e.g., logging and wizards) or a frontend for running TGG integrations.

16.1 STORY DRIVEN MODELING

The meta-CASE tool eMoflon uses three different metamodeling technologies: (i) It is possible to specify Ecore metamodels to establish domain-specific languages (i.e., their static semantics). (ii) Unidirectional graph transformations can be used to specify the behavior of elements by means of method implementations (i.e., their dynamic semantics). (iii) Bidirectional graph transformation (i.e., TGGs) can be used to specify an integration between different metamodels. All these components are further used as a source for generating Java

1 eMoflon can be downloaded from www.emoflon.org together with an extensive tutorial and exemplary application data.

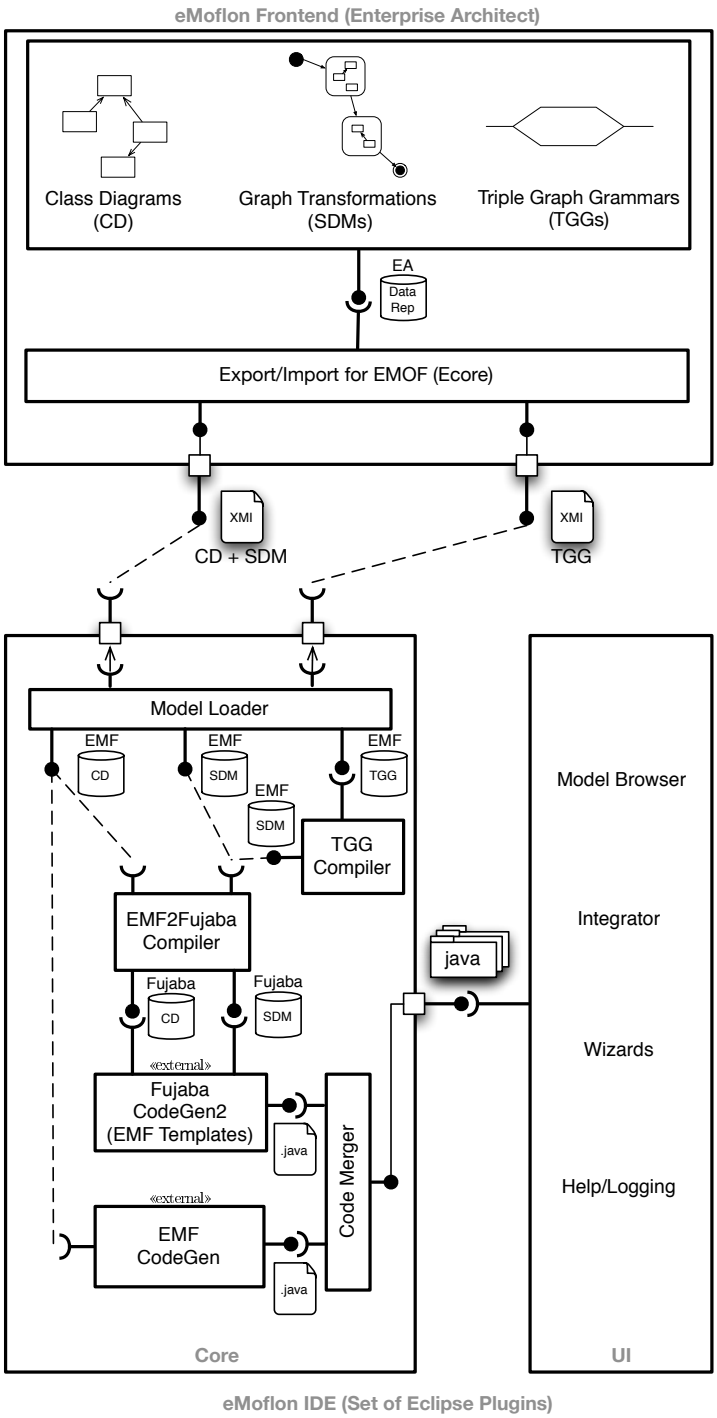


Figure 79: System overview of eMoflon [ALPS11]

code in order to actually execute the specification. In the following, we will have a closer look at specifying the behavior as this has been used to define large parts of eMoflon itself. This concept is referenced as bootstrapping.

eMoflon uses the approach of *Story Driven Modeling* (SDM), which is a composition of declarative graph transformations and imperative control flow definitions [FNTZ00, vDRH+11]. It is, therefore, possible to define the behavior of a method via expressing model modifications with standard graph transformation rules as presented in Chap. 3 and additionally embed these rules in control flow elements which support statements, loops and conditions. Figure 80 depicts such a control flow.

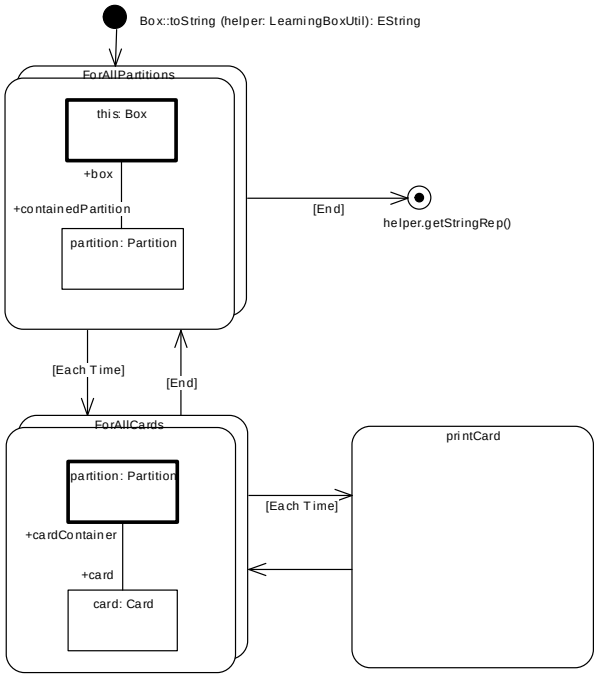


Figure 80: SDM control flow specification with loops

The method entry on top is depicted by a filled black circle. From this element, an edge goes to a double black-bordered box, which denotes a for-loop. This loop iterates over all `partitions` of a given `box`. For each iteration, another for-loop is started iterating over all `cards` of a `partition`. The empty box `printCard` denotes an omitted graph transformation which is not displayed to improve readability. Having iterated over all `partitions`, the method terminates with the bordered black dot in the upper right quarter of Fig. 80.

Another control flow component is depicted in Fig. 81. Here, a method starts checking if the attribute `back` of a `card` object has been set to the value of `guessed`. If this is not the case, the control flow follows the edge annotated with `Failure`. If the values are equal, the

control flow follows the edge annotated with `Success` and triggers the next condition checking.

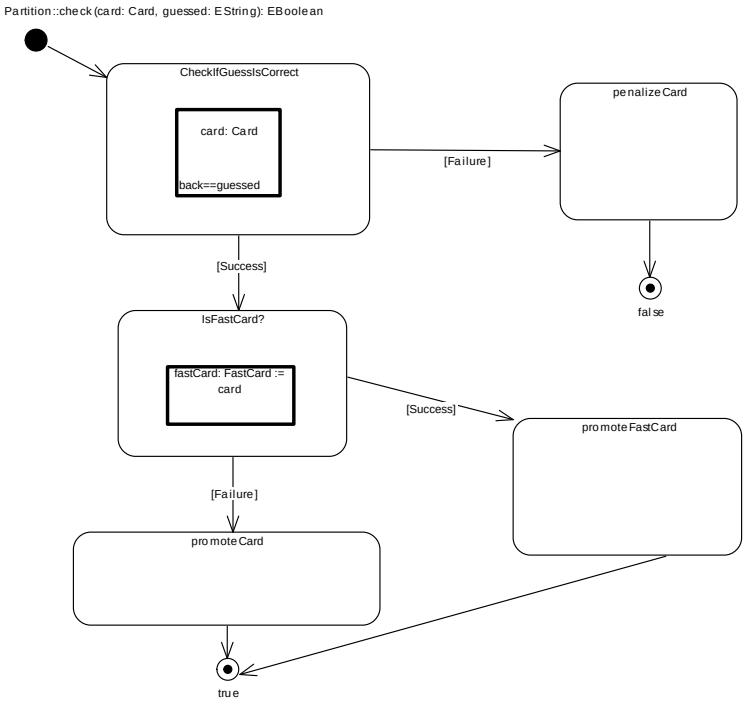


Figure 81: SDM control flow specification with conditions

So far, no actual model transformation rule has been presented. Hence, Fig. 82 depicts a rule that moves a `FastCard` from a `Partition` to another `Partition` and, therefore, denotes an actual model transformation.

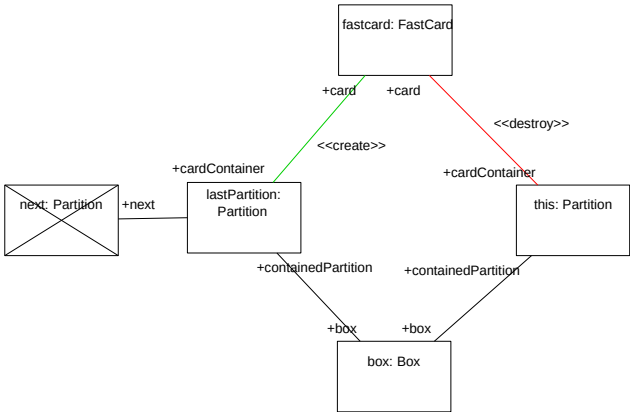


Figure 82: SDM model transformation pattern

The rule expects the objects `fastcard` and `this`, which denotes the `Partition` in which this method is executed, to be already found (thick bordered). The objects `box` and `lastPartition` are searched. The crossed-out object to the left represents a so-called *Negative Ap-*

plication Condition (NAC). Such a NAC is used to state that a certain condition must not hold in order to apply the overall rule. Therefore, if all elements are found, two changes are applied to the model: (i) A new connection between `lastPartition` and `fastcard` is established (green edge with «create» markup). (ii) An existing connection between `this` and `fastcard` will be removed (red edge with «destroy» markup). Furthermore, it is possible to create and delete objects and to adjust attribute values (not shown here).

16.2 THE ADDED VALUE OF GRAPH TRANSFORMATIONS

Graph transformation has been a research topic for a long time. For about 30 to 40 years, graph transformation approaches have been developed and improved. Typically, two arguments are used to emphasize the benefits of graph transformation: (i) The level of abstraction is considerably higher than compared to a standard general purpose programming language implementation and (ii) due to a graphical representation, these specifications are easy to read and understand [BWW11].

Nevertheless, it seems that graph transformations are rarely used in real-world application scenarios. The question why this is the case has been researched by Buchmann et al. [BWW11] exemplary on a basis of two large programs, implemented with graph transformations (in both cases SDM was used). They discovered that graph transformation in large projects are often used to express small (trivial) model modifications in highly procedural processes. Together with concise metrics, they deduced that model transformation seems to be more or less standard Java programming on a higher-level only.

In the following discussions, we will use these metrics to check whether our implementation is also programming on higher-level or if we were able to actually use graph transformations.

Of course, these findings are sobering. But from our point of view, the use of graph transformation to implement eMoflon was the right choice for the following reasons:

- Structural and behavioral specifications can be reviewed and edited in one place. Typically, users have to edit class diagrams (structural specifications) independently from behavioral specifications (i.e., source code). With our approach, we are able to significantly reduce the number of program switches and, hence, increase productivity.
- The whole model-driven specification is a good documentation. Regardless of technical details, the documentation can be understood and this documentation focuses on relevant aspects only.

- Introducing new aspects to existing models (either structural or behavioral) can be achieved easily as our user interface in EA supports the reuse of existing elements.
- Using graph transformation demands for a certain amount of discipline. Methods have to be concisely structured and data structures must be set up appropriately. These imperatives increase the quality of the resulting code artifacts because common concepts are used frequently and, therefore, improve understandability and maintainability.

Altogether, graph transformation seems to be the right choice for implementing complex data manipulation programs as long as these data structures have a graph-like structure. Of course it is necessary to provide engineers with fall-back solutions to embed handwritten code for such cases where graph transformations can hardly be used (e.g., establishing a connection to a database). Nevertheless, if you have the opportunity (as we had) to use graphs as basic data structures it is definitely possible to intensively and productively use graph transformations for implementing large projects. And finally, we will see in our bootstrap chapter (cf. Chap. 18) that the criticism, model transformations are used for almost trivial (i.e., small) patterns only, is not true for all application scenarios.

17

TGG CONTROL ALGORITHM

In this chapter, the implementation with model transformations of the context-driven control algorithm of [KLKS10] is presented. This algorithm, as presented in Sect. 4.6, operates in different phases. To regard readability and space consumption we selected three major steps in the algorithm to highlight how the implementation with SDM of the algorithm looks like: (i) The overall transformation process is started, which invokes a transformation on each untransformed node in the input model. (ii) Processing each untransformed element recursively transforms all potentially required context nodes. (iii) When all context elements have been processed, a rule to transform the regarded node has to be selected and applied.

17.1 SDM SPECIFICATION

Step (i): The algorithm of [KLKS10] supports batch model transformations. The basic idea of this algorithm is to process an input model in a context-driven manner. Therefore, the overall process of the algorithm starts randomly selecting one node of the input model and transforms this node via calling method TRANSLATE(`element`). Figure 83 depicts this process. Here, after an initialization phase, where basic preconditions are checked, a for-loop iterates over all untransformed elements.[1] The algorithm itself is represented by a `Translator` instance that maintains two sets of elements: a set of processed nodes and a set of unprocessed nodes. The crossed-out connection between `this` and `element` denotes that the for-loop only iterates over elements that are not in the set of processed nodes. Every time such an element is encountered, method TRANSLATE(`element`) is called. The for-loop iterates once over all unprocessed nodes. This is sufficient as the algorithm recursively calls TRANSLATE(`element`) for every element that is supposed to be processed in order to translate the recent node.

Step (ii): The actual transformation process takes place in method TRANSLATE(`EObject`), depicted in Fig. 84. After checking certain parameters, a set of candidate rules is retrieved. This set denotes all rules that are on a type level applicable to the element given as an argument to this method. Unfortunately, this does not yet guarantee

1 Note that in the algorithm implementation the term "translate" was chosen instead of "transform". We decided not to adjust the SDM specification but to use these terms synonymously in this chapter.

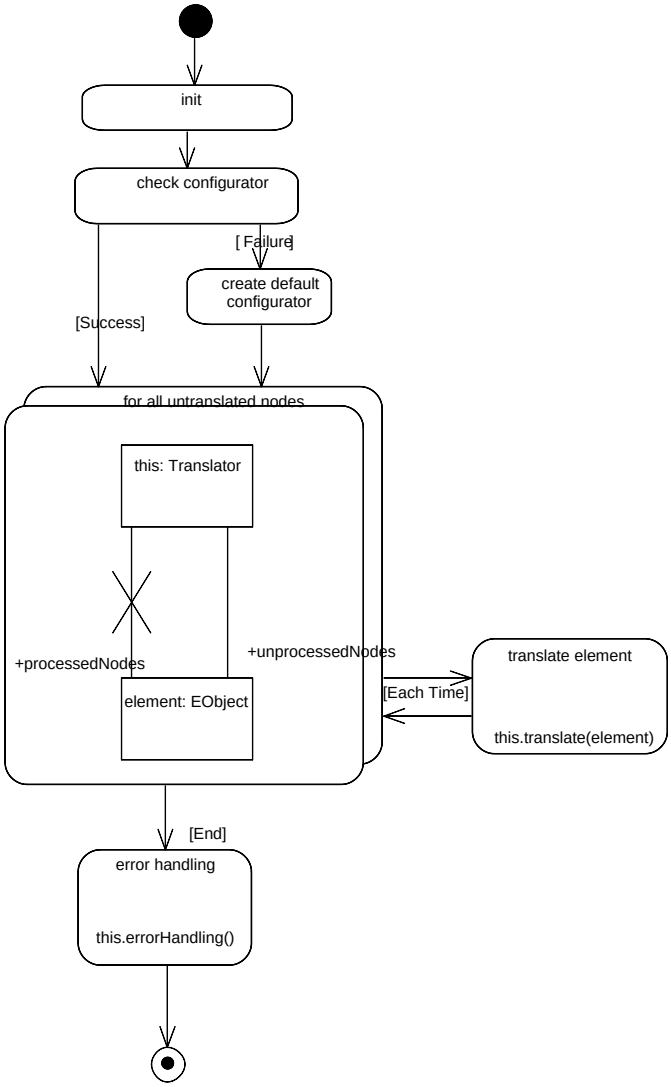


Figure 83: The outer transformation loop of the TGG control algorithm

that the node is actually processable by this rule. Hence, further actions have to be applied (i.e., match completion, DEC check, context element transformation). Especially the context element transformation is interesting as it denotes the starting point of the recursive transformation process.

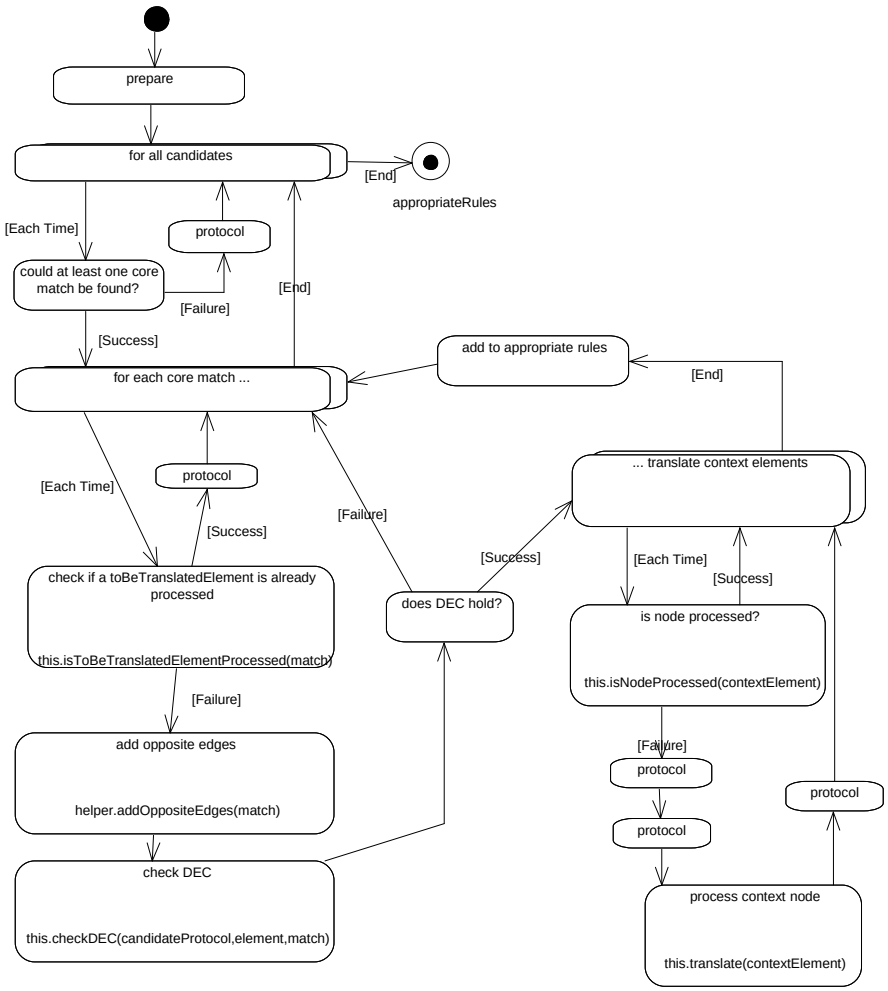


Figure 84: Transformation process of a single model element with recursive context transformation

Step (iii): When all context elements have been transformed, method collectAppropriateRules() (depicted in Fig. 85) is invoked, where the current element shall be transformed. This method works with the set of previously determined candidate rules. Each time, the method tries to complete the match not only in the input but also output domain and correspondence domain. If such a match could be found, the control algorithm checks whether (negative) application conditions and constraints are fulfilled and if this is the case, the rule is actually applied. If this is not the case, the algorithm checks whether another match exists that can be used, or if another rule is applicable to to this element.

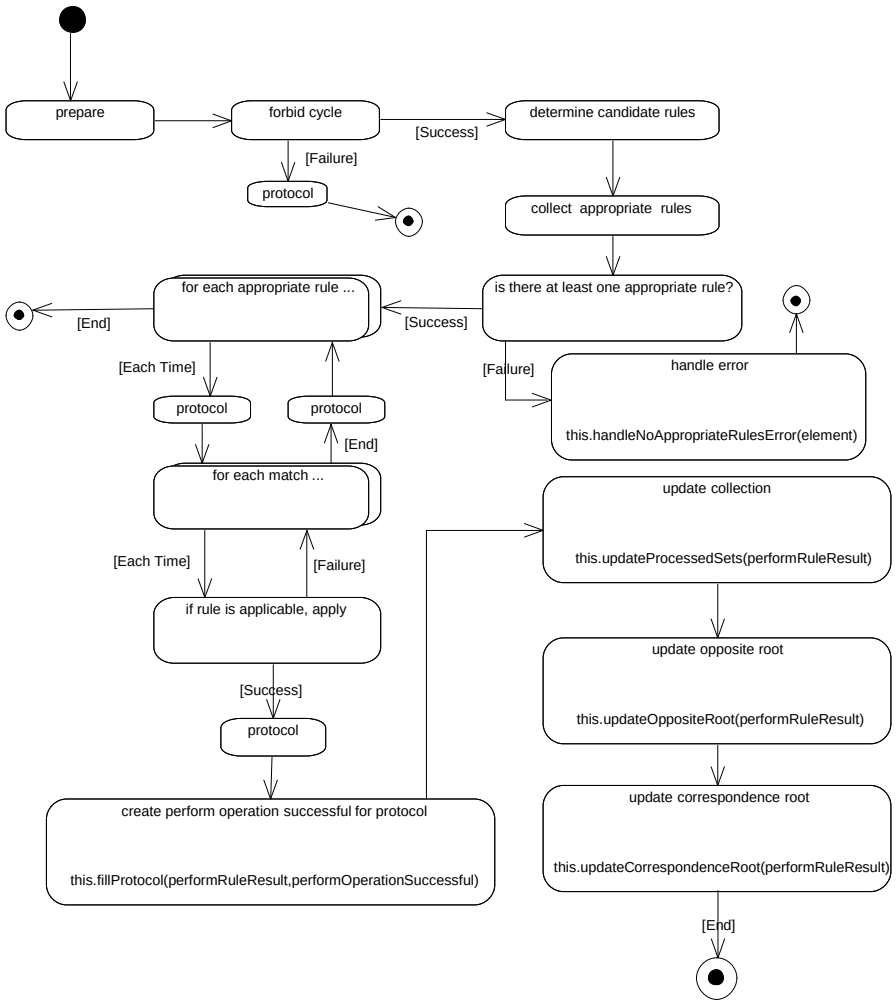


Figure 85: Collecting appropriate rules and applying a rule to a model element

If one rule could be applied, all corresponding sets have to be updated and the algorithm may safely quit this transformation step. The outer loop (cf. Fig. 83) now carries on selecting unprocessed nodes and repeats the transformation process from (i).

17.2 RUNTIME EVALUATION

Benchmarking model transformations has already been subject of publications such as [VSV05, GK07]. Typically one or more exemplary scenarios are selected to measure the runtime complexity of the transformation approach compared to the size of the input model. This exactly is what we are going to do in this section as well. Please note that our small benchmark is by no means complete or claims to be a representative of practical TGGs. This benchmark is intended to provide us with a feeling of how model transformations with TGGs perform.

The TGG which was used in this benchmark consists of four TGG rules depicted in Fig. 86 and an appropriate type graph triple (not depicted).

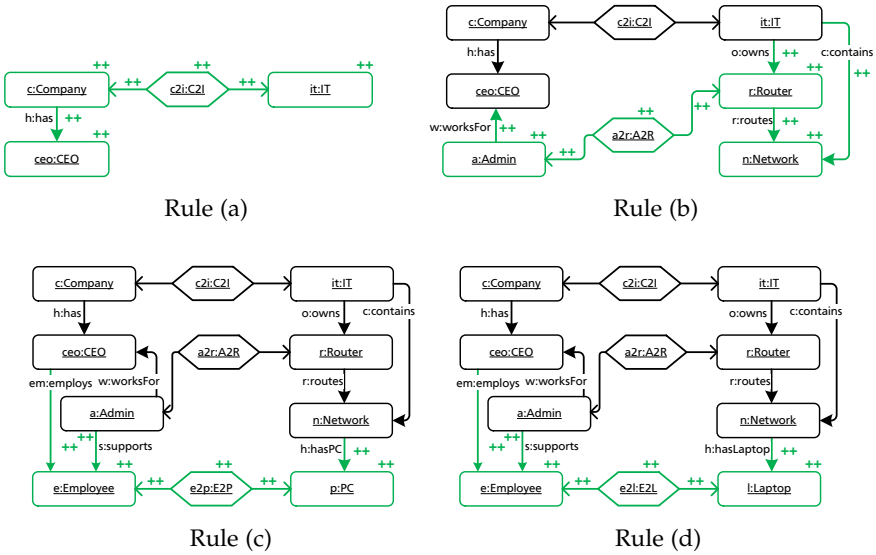


Figure 86: Set of TGG Rules used for the evaluation of the TGG control algorithm

These four rules build up an integrated company and IT structure simultaneously. Rule (a) creates the root elements of the models (a `Company` with a `CEO` and a corresponding `IT`), while Rule (b) appends additional elements (an `Admin` and a corresponding `Router` with the controlled `Network`). Rules (c) and (d) extend the models with an `Employee`, who can choose a `PC` or a `Laptop`. Rules (c) and (d)

denote the non-functional part of our TGG specification as here different rules are applicable to the same element.

In the following we will discuss how the previously described implementation of the context-driven TGG approach behaves in combination with the rules depicted above. The expected result is that the runtime grows polynomially with n^k as an upper bound. All performance tests have been applied on randomly created input `Company` models of a specific size. This random model generation process and the actual benchmark were achieved by using the model benchmarking framework of [HLG+12]. The algorithm was tested in a black-box manner: The input model is passed to the algorithm, which initializes helper data structures and wrappers, and then applies the actual model transformation. Hence, this initialization time is also part of our measured times. All tests have been repeated for 20 times with each input model on a Dual-Core Pentium E6750@2.66GHz with 8GB of RAM. The 64bit virtual machine used was configured with 2GB of heap space. Figure 87 depicts the retrieved results.

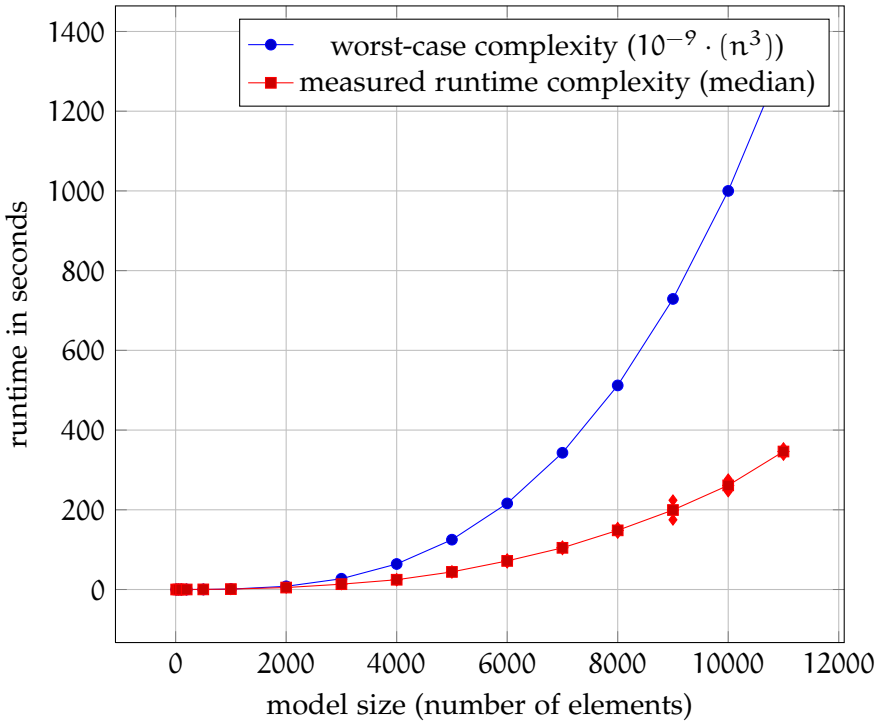


Figure 87: Measured vs. worst-case runtime complexity of integrating instances of the running example with different model sizes

The depicted measured values represent the median runtime of 20 test runs per model size (except for 10,000 and 11,000 elements as these model sizes caused a memory out of bounds exception as described below). We selected the median as it is more robust regarding extreme values. Nevertheless, these values are still covered by the interval around each data point.

The graph depicts next to the measured runtime values a curve of the expected worst-case complexity. As our TGG rules (cf. Fig. 86) specify patterns where at most three elements have to be matched, the expected worst-case complexity is denoted by $\mathcal{O}(n^3)$. This curve was scaled by multiplying 10^{-9} to fit into the same scale with the measured results without compromising its characteristics.

Regarding Fig. 87 we can clearly see that the measured results are all equal or below the expected worst-case complexity. As a conclusion, we can clearly say that the implementation behaves as expected.

At least this is true until a cumulative number of about 180,000 elements have been loaded into memory in total. As we repeated our benchmark 20 times for each model size, this effect did not appear until we reached the model with 10,000 elements. Benchmark runs 1 to 18 could be performed within about 180 seconds, while runs 19 and 20 run with a duration of about 1,000 seconds (note that the model didn't change!). A similar behavior could be observed for benchmarks with a model of size 11,000. This time, the runtime explosion already happened after 16 runs. Runs 1 to 16 could be accomplished within about 360 seconds, while runs 17 to 20 required from 1,000 to 2,000 seconds. Obviously, some memory leak has been discovered which forces the virtual machine to run the garbage collector. This definitely needs further investigation. Finally, a benchmark run with 20,000 elements emphasized this finding, as the runtime explosion could be determined already after 9 runs.

17.3 EVALUATION OF OUR IMPLEMENTATION

The algorithm implemented in our tool eMoflon has been formally developed and proven first and was implemented afterwards. This consequent use of formalisms helped us understanding the algorithm. Furthermore, the use of graph transformations increased the level of documentation massively (compared to the preceding algorithm implementation) as now, for the very first time, documentation and implementation have been achieved in one step. Typical agile development practices such as pair programming (or in this case "pair modeling") additionally helped to improve efficiency and quality of the developed product.

The following Figs. 88 and 89 compare our TGG control algorithm implementation with two exemplary programs of [BWW11]. These two exemplary implementations are a version control system (left-hand side) and the implementation of a code generator for SDM specifications named Codegen2 [GSR05] (center). Figure 88 provides values regarding the size of the modeled programs considering properties such as the number of packages, classes and the overall number of variables in SDMs.

Model element	Specification #1	Specification #2	TGG algorithm
Number of Packages	68	18	20
Total number of Classes	175	162	121
Number of abstract Classes	18	28	13
Number of Interfaces	32	10	0
Total number of Attributes	177	181	81
Total number of Methods	650	443	45
Number of Generalizations	220	247	69
Number of Associations	148	166	180
Number of Story Diagrams	540	339	13
Number of Story Patterns	988	850	94
Total number of Objects	1688	1997	260
Number of Negative Objects	42	22	3
Number of Set Objects	25	9	0
Total number of Links	725	1121	168
Total number of Paths	13	7	0
Number of Statement Activities	264	64	22
Number of Collab Calls	1183	711	0
Number of ForEach Activities	27	88	23

Figure 88: Comparison of general values of exemplary SDM implementations according to [BWW11] and of our algorithm implementation

The most significant ratios are computed in Fig. 89.

Significant numbers	Specification #1	Specification #2	TGG algorithm
Classes per Package	2.57	9.00	6.05
Attributes per Class	1.01	1.12	0.67
Methods per Class	3.71	2.73	0.37
Patterns per Story Diagram	1.83	2.51	7.23
Objects per Pattern	1.71	2.35	2.77
Links per Pattern	0.73	1.32	1.79
Statement Activities per Story D.	0.27	0.08	1.69
Collab Calls per Story Pattern	1.20	0.84	0.00
Collab Calls per Story Diagram	2.19	2.10	0.00

Figure 89: Comparison of computed ratios of exemplary SDM implementations according to [BWW11] and of our algorithm implementation

Considering the exemplary implementations, we can see that in average one attribute exists per class and each class has about three methods. Regarding the complexity of modeled method implementations with SDM, we found that 1.83 and respectively, 2.51 patterns exist in average per method. Furthermore, in each of these patterns consists of 1.71, respectively 2.35 object variables and 0.73 / 1.32 link variables. This means that these patterns are typically small and only weakly interconnected (i.e., only few edges are used within the pattern). Interesting values are the numbers of collab calls and statement activities as they both denote a fallback solution for using Java inside a model transformation. These values are indicators of how frequent

Java code has been called and, therefore, only control flow has been modeled.

Regarding our algorithm implementation the overall size of the specification cannot be compared to the size of the exemplary projects as the exemplary projects are built of more packages and classes but interesting values can, nevertheless, be aggregated. Consider the value for patterns per story diagram: 7.23. This value is more than three-times higher than the values of the exemplary projects and represents the complexity of the control flow within method implementations. The number of objects per pattern is comparable to the other implementations with about 2.76. Interesting to see is that the number of links (i.e., edges) per pattern (1.79) is significantly higher, which means that our specified patterns are more connected. Finally, consider the values for statement activities and collab calls. As we do not support collab calls in eMoflon, fallback solutions have to be modeled via statement activities. Interpreting these values all overall, it seems that using model transformation and SDM in special for modeling the behavior of a control algorithm is a suitable approach. This is especially interesting if we reconsider the fact that SDM is intended to be used to specify model transformations and not control algorithms for model transformations.

18

OPERATIONAL RULE DERIVATION

As explained previously, the main components of a TGG implementation are (i) a given TGG specification, (ii) a control algorithm and (iii) a compiler for deriving operational rules (i.e., unidirectional model transformations). This chapter describes how such operational rules can be derived with model transformations as implemented in our tool eMoflon.

18.1 SDM SPECIFICATION

The formal theory of deriving operational rules from TGG rules has been discussed and expanded in Sect. 4.1 and Sect. 9.2. In the following this process is presented for deriving a forward rule from a given TGG rule.

The overall process is depicted in Fig. 90. Deriving operational rules requires processing all specified TGG rules, which is achieved by iterating over all TGG rules. The first loop produces rule specific transformation actions which will be applied when a certain context defined by the TGG rule has been successfully matched. The second loop iterates over all `object variables` within the given TGG and generates variable specific operational rule parts. An `object variable` (short `OV`) represents any element in a rule that represents a typed element of a pattern to be matched (i.e., a node).

A restriction of the original publication of [KLKS10] was that only one create element was allowed per domain and rule. Otherwise it was unclear how to apply the same rule to transform different elements. Our solution to this issue was to derive a bunch of different operational rules: one per create element per TGG rule. The benefit is that each of these derived rules will apply the same action to the graph triple, but starts the actual pattern matching process at different elements. Once, the context has been matched successfully, the rest of the operational rules, i.e., actually applying the model transformation is identical. Hence, this starting element denotes the point of variation regarding the different operational rules per TGG rule.

Considering again the method depicted in Fig. 90, the derivation process checks whether a forward or backward transformation has to be derived. If the `OV` under consideration is placed in the source domain then a forward transformation has to be derived and vice versa. At next, a so-called pattern for the DEC check is computed

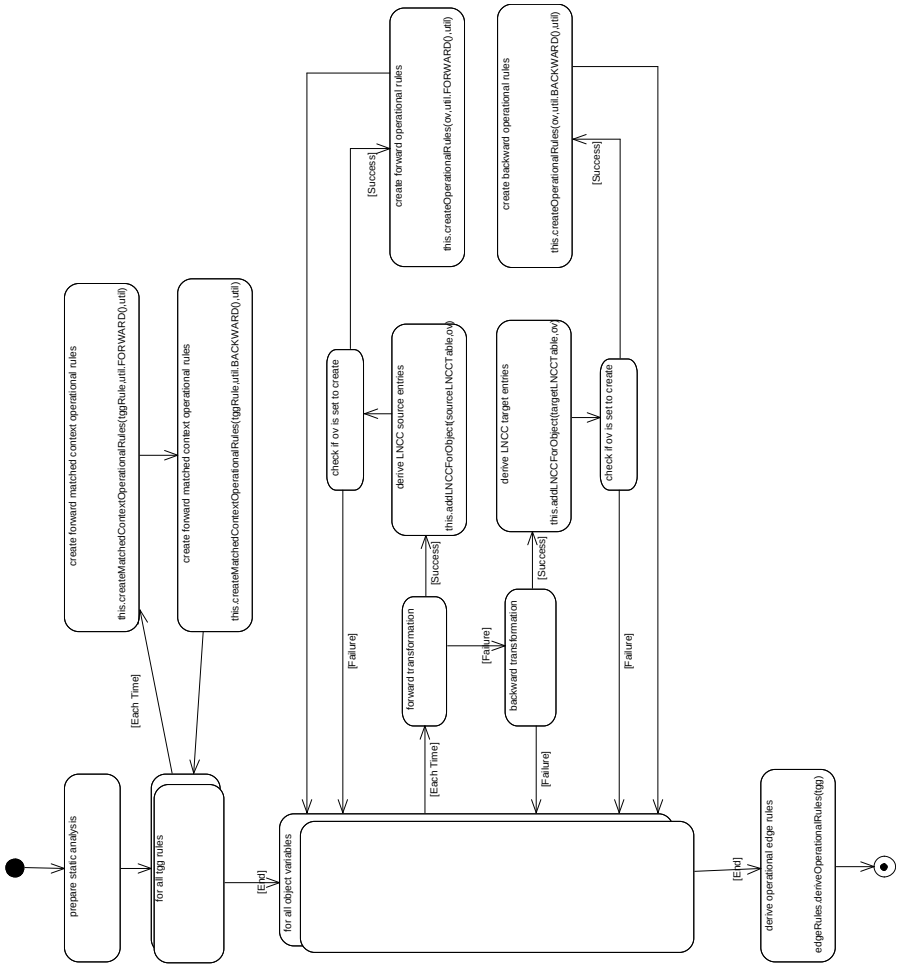


Figure 90: Overall compilation process of deriving operational rules from TGG rules

which is necessary discover dangling edges at runtime. If the `OV` does not represent a created element in the TGG rule, the loop skips this OV. Otherwise, an appropriate operational rule has to be derived.

Figure 91 depicts the actual deriving process. This process starts with an initialization phase where technical details such as creating `eOperations`, adding appropriate parameters and some more are performed. Next, the correct entry node is set and afterwards the original TGG rule is cloned. This is a sufficient intermediate step as now only all created elements of one domain (e.g., the source domain if a forward rule is derived) have to be set to context, which means they have to be matched. Setting the entry node is necessary to define in the derived operational rule which element is already found and, hence, the starting point for the matching process of this rule.

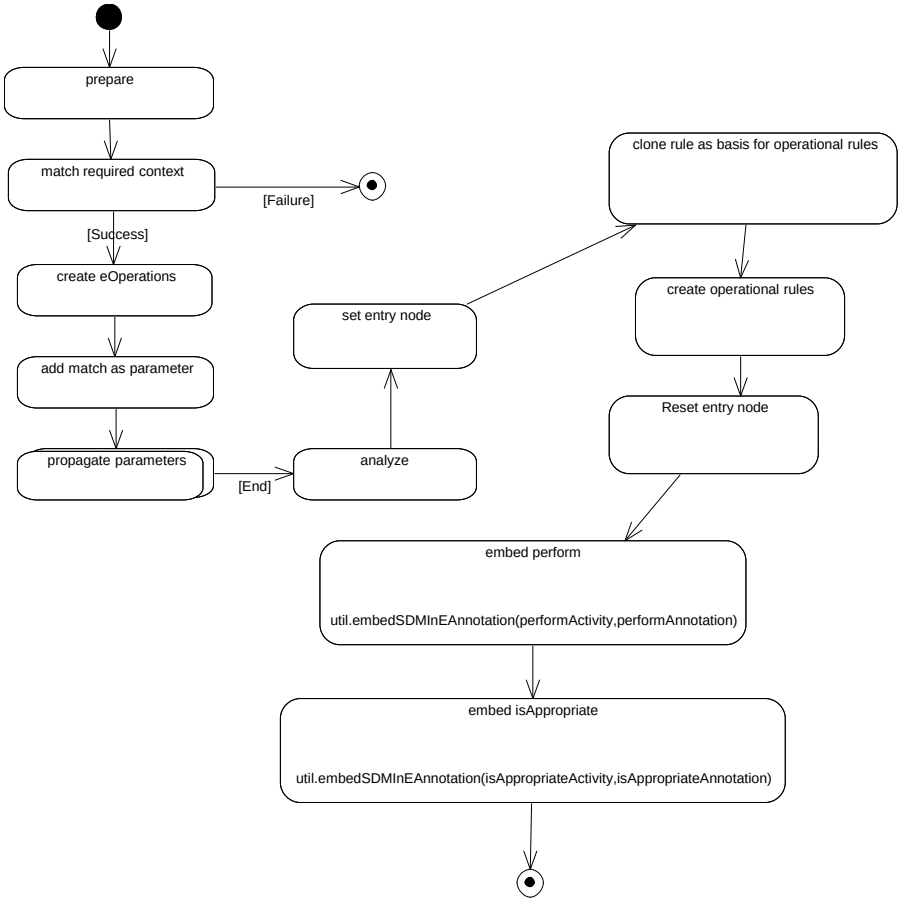


Figure 91: Process of parsing each TGG rule

The process of creating the perform operation is depicted in Fig. 92. Here, a loop iterates over all `OVs` of a previously cloned rule. Whenever an `OV` of the input domain is encountered, this `OV` has to be stored as an required element. Additionally, if this `OV` was set to create, this has to be adjusted to context. Next, a further loop iterates over all `link variables` (`LVs`), which denote variables `eReferences` within

TGG rules. Here, basically the same has to happen: whenever a created `LV` is placed in the input domain this has to be adjusted to match as context. Thus, additional patterns and nodes are created within the derived operational rule, before the SDM structure is completed.

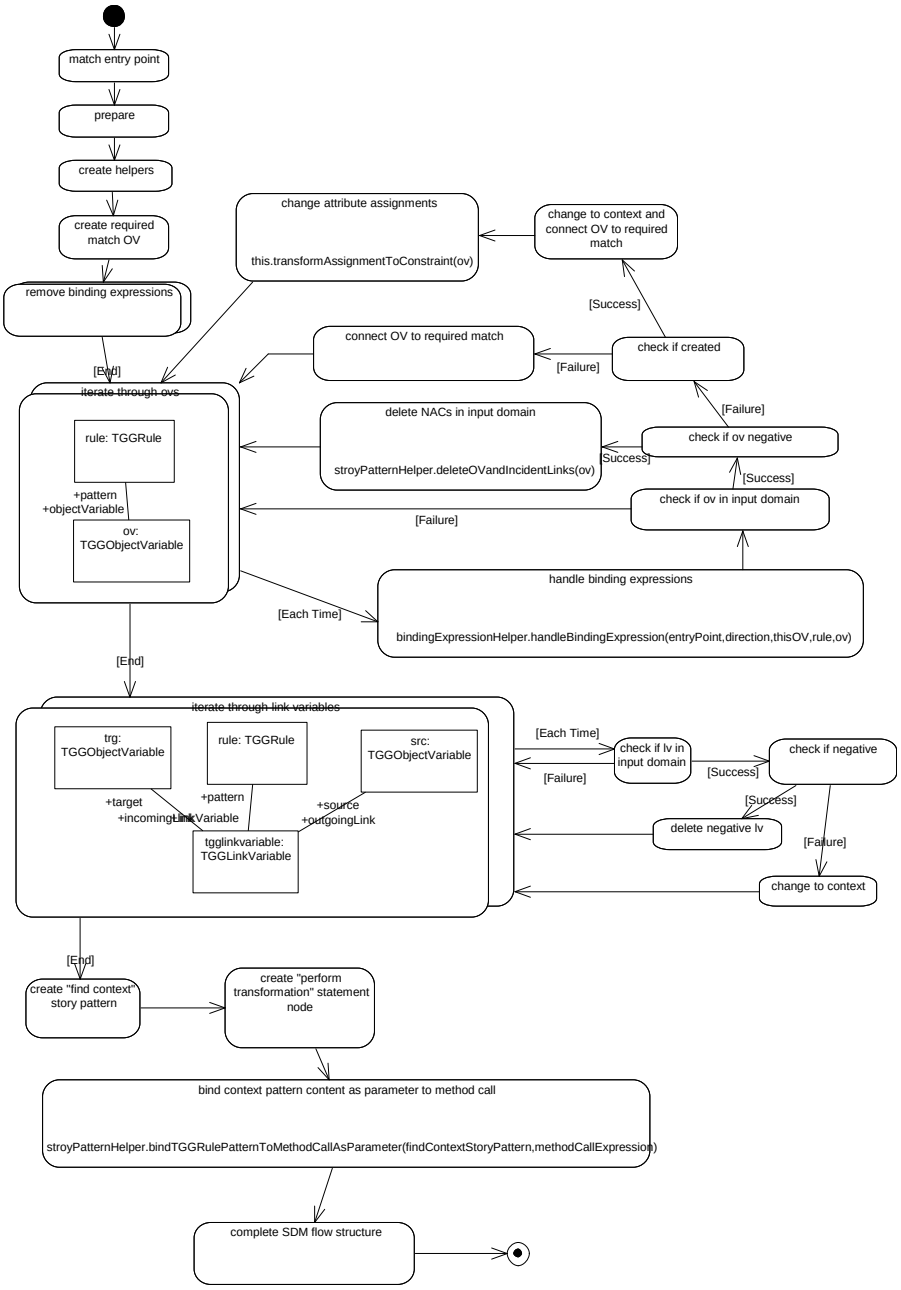


Figure 92: Process of building a concrete perform operation (i.e., a forward or backward rule)

Figure 93 depicts the action of embedding all previously derived parts of the operational rule into an overall method declaration. Here, the various parts of the operational rule are appended via `story nodes` to a `story activity`. `Activity edges` are created between these `story nodes` to denote the control flow.

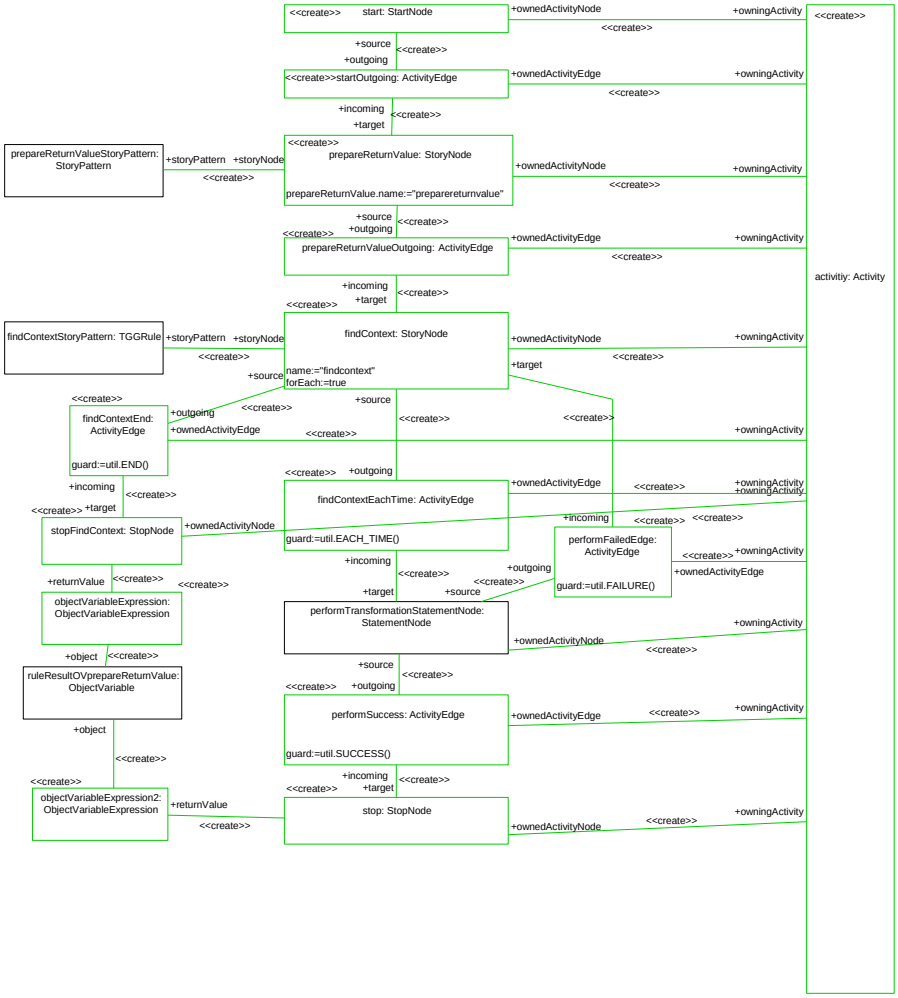


Figure 93: Model transformation to assemble a complete operational rule

Note that the whole compilation process does not produce any line of code but only `story activities`. These activities denote unidirectional operational rules as described in Def. 13 and are further used by a code generation engine to produce Java code.

Example

Consider the TGG rule of our running example, depicted in Fig. 94 (i.e., Rule (a)). This rule states that an `Admin` will be employed and, therefore, assigned to a `Router` and a `Network` for maintenance purposes.

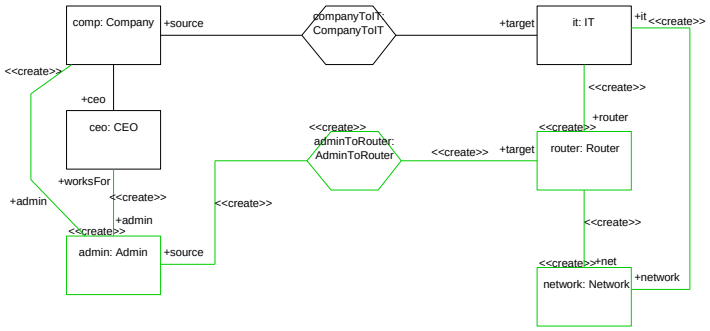


Figure 94: Exemplary Rule (b) (cf. Fig. 86) as specified in eMoflon (integrates an `Admin` and a `Router` object)

Therefore, the derived operational rule for forward transforming a given `Admin` object has to be able to search for the appropriate match, i.e., check whether the given `Admin` object is connected to a `CEO` and a `Company` and, furthermore, if the latter is also connected via a correspondence link to an `IT` object. Such a rule is depicted in Fig. 95.

The important point is that various possible matches may exist that fulfil the requirements and, therefore, the operational rule has to apply the actual model transformation to one of those matches. For this reason, a loop iterates over all possible matches. Whenever an appropriate match is found, the model transformation depicted in Fig. 96 is called. As soon as this method succeeded once, the rule application exits as the unprocessed `Admin` object has now been transformed.

The actual model transformation which modifies the correspondence and target domain is depicted in Fig. 96. Here, appropriate `Router` and `Network` objects are created and connected to the `Admin` object of the source domain.

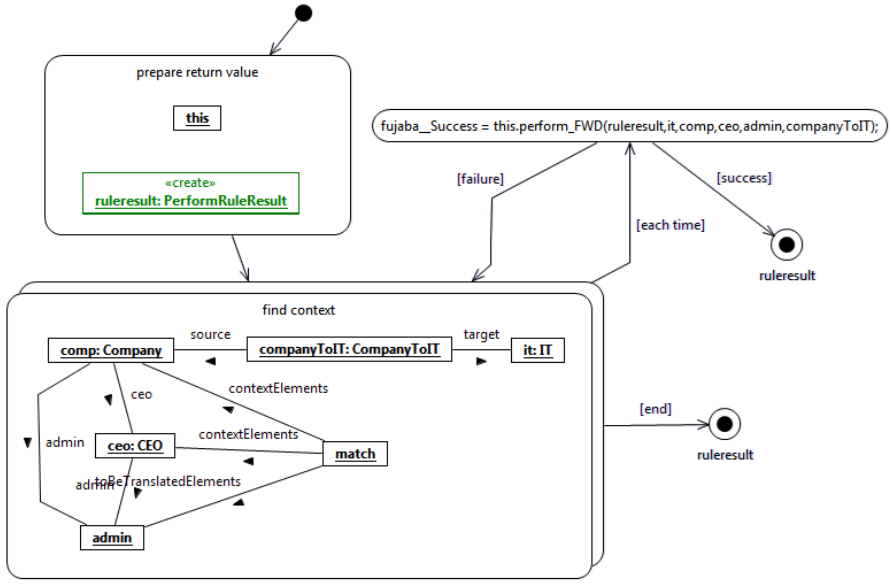


Figure 95: Excerpt of the derived operational forward rule searching for all potential matches

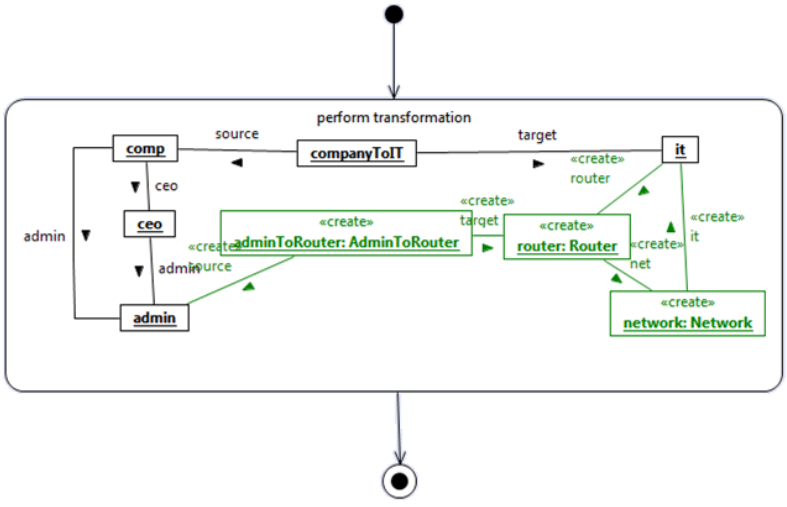


Figure 96: Excerpt of the derived operational forward rule applying an actual model change to the target domain

18.2 EVALUATION OF OUR IMPLEMENTATION

To evaluate our approach from a quantitative point of view, we consider again the contribution of Buchmann et al. [BWW11] and compare also this implementation with the two exemplary implementations of their publication (cf. Figs. 97 and 98).

Model element	Specification #1	Specification #2	TGG compiler
Number of Packages	68	18	23
Total number of Classes	175	162	131
Number of abstract Classes	18	28	13
Number of Interfaces	32	10	0
Total number of Attributes	177	181	81
Total number of Methods	650	443	59
Number of Generalizations	220	247	59
Number of Associations	148	166	189
Number of Story Diagrams	540	339	27
Number of Story Patterns	988	850	187
Total number of Objects	1688	1997	739
Number of Negative Objects	42	22	0
Number of Set Objects	25	9	0
Total number of Links	725	1121	611
Total number of Paths	13	7	0
Number of Statement Activities	264	64	35
Number of Collab Calls	1183	711	0
Number of ForEach Activities	27	88	35

Figure 97: Comparison of general values of exemplary SDM implementations according to [BWW11] and of our compiler implementation

The most significant ratios are computed in Fig. 89.

Significant numbers	Specification #1	Specification #2	TGG compiler
Classes per Package	2.57	9.00	5.70
Attributes per Class	1.01	1.12	0.62
Methods per Class	3.71	2.73	0.45
Patterns per Story Diagram	1.83	2.51	6.93
Objects per Pattern	1.71	2.35	3.95
Links per Pattern	0.73	1.32	3.27
Statement Activities	0.27	0.08	1.30
Collab Calls per Story Pattern	1.20	0.84	0.00
Collab Calls per Story Diagram	2.19	2.10	0.00

Figure 98: Comparison of computed ratios of exemplary SDM implementations according to [BWW11] and of our compiler implementation

Analyzing the data of these tables, we see that the graph transformation implementation of our compilation process consists of more than one hundred classes and nearly sixty methods.[1] Thus, each method implementation consists of 6.93 patterns in average, which reflects

1 The TGG metamodel and supplementary data structures are also regarded by these numbers (influences number of packages, classes, attributes, etc.). Nevertheless, the values regarding graph transformations are not distorted as these additional data structures did not contain additional graph transformations.

the size of the control flow. Furthermore, each pattern consists of about 3.95 object variables and 3.27 link variables. Hence, the here implemented patterns exceed the size of the exemplary model transformation and denote, therefore, more complex patterns.

From a qualitative point of view, deriving operational rules with model transformations denotes a (semi-)bootstrap, which shows that model transformations are an appropriate technique to cope with such compilation processes. As a bootstrap is the technique of building a product by using the product itself, this implementation is a semi-bootstrap only for the reason that we did not use TGGs to derive operational rules. Considering our scenario, two reasons lead to this decision: (i) It is not a requirement to realize the operational rule derivation via bidirectional transformations; (ii) It remains unclear if TGGs are a applicable technique in such a scenario. As some technical details and intermediate data structures have to be used throughout the transformation, it was necessary to represent these elements in the transformation also. Such technical elements often have no semantic equivalent in the opposite domain and, hence, it is hard to build an appropriate bidirectional rule specification.

As future work, we identified two major areas where extensions and improvements are necessary:

- Regarding our incremental model synchronization approach, it is necessary to derive also inverse forward and backward rules from a given TGG rule. This can be achieved similarly compared to the derivation process described in this chapter. Nevertheless, new questions may arise regarding the uniqueness of the elements that shall be removed. Deleting elements in the opposite domain (e.g., the target domain in case of a forward transformation) can be compared to processing the same elements in the opposite domain as input. In the latter case, problems are caused when two rules are applicable to a common subset only. Hence, as future work not only the derivation process has to be extended but also the control algorithm will have to check whether two or more inverse rules are applicable at the same elements and, therefore, test if dangling edge conditions are fulfilled in either case. For this reason it may be necessary to pass additional information to the operational rule or allow for user input.
- Another open issue is that the derivation process suffers from high memory and time consumption. Deriving a set of operational rules takes from a few seconds up to minutes regarding the size of the TGG. Profiling showed that internally, large collections are implemented with the generic Ecore data structure `EList` and, therefore, claim most of the actual runtime with adding elements to those lists and/or updating the lists after changes. A promising idea is to exchange the underlying data structure with a more

efficient data structure one. Such a data structure may, for example, cache frequently retrieved elements or may be balanced to allow for equal access times regardless which element has to be retrieved. Another possibility could be to reduce the number of derived operational rules. As discussed previously, we derive one operational rule for each create element. This is necessary as the context-driven control algorithm selects randomly an element to be processed and, therefore, needs to know which rule can be applied. For this reason, numerous operational rules have to be created. In contrast, our precedence-driven control algorithm utilizes a precedence graph for determining the traversal order. Within this precedence graph, equivalence classes denote a set of nodes that can be processed together (cf. Chap. 6). An optimized approach could determine specific nodes that act as a representative. With these representatives, the control algorithm is still able to determine appropriate rules which will definitely process all other elements in the equivalence class. The results will be less operational rules to compile and, hence, an improved compilation runtime performance and secondly an improved runtime performance as (probably) only these representative nodes have to be regarded when building and maintaining the precedence graph.

Part VI

CONCLUSION AND FUTURE WORK

CONCLUSION AND FUTURE WORK

The aim of this thesis was to develop an approach for bidirectional model synchronization. Therefore, the hypothesis reads as follows:

Hypothesis
The aim of this thesis is to show that incremental model synchronization can be achieved with TGGs efficiently, while retaining as much information as possible. Furthermore, this solution has to be efficient and to guarantee the formal properties introduced in [Sch95, SK08].

To achieve this, the following was contributed:

1. We proposed a so-called **precedence analysis** in Chap. 6, which builds the fundament of efficient and information preserving model synchronization. With this **static analysis**, it is possible to (i) analyze a TGG specification statically to **detect specification errors** and (ii) utilize statically retrieved information to **define a partial order** of a model at runtime.
2. We developed in Chap. 7 a novel **batch transformation algorithm**, which uses information from the precedence analysis to determine an appropriate and efficient traversal order through the input model. We proved that this **algorithm complies with the required formal properties** and **is efficient** as its runtime complexity scales polynomially with the size of the model.
3. We discussed formal aspects of propagating deletions in Chap. 9 and **extended the TGG theory** with a formal concept of deriving rules that delete elements (i.e., **inverse rules**). Such inverse rules are necessary to **revoke the effects** of previous rule applications.
4. The **incremental model synchronization algorithm** that uses all the previously introduced concepts was presented in Chap. 10. The algorithm operates in **three phases**: (i) propagate deletions (ii) prepare affected elements of additions, and (iii) transform all unprocessed model elements. Again, this algorithm **complies to all formal properties** and its polynomial runtime complexity **scales provably with the number of model changes and affected elements**. Thus, the control algorithm **retains as much information as possible**.
5. In Chaps. 12–14 **improvements of our approach** provably guarantee to (i) reduce the number of elements to be processed during a model synchronization and, therefore, **improve efficiency**

and retain even more information, (ii) **extend expressiveness** of TGGs, and (iii) provide **static checks for potential specification errors**.

Additionally, in Chaps. 17 and 18 we reviewed implementation details of the currently implemented TGG approach that has been extended by this thesis. By applying metrics for model transformations and evaluating the runtime complexity of the control algorithm, we showed empirically that the theory of TGGs in combination with our implementation is capable of complex model transformation tasks.

From a theoretical point of view, all challenges described in Chap. 1 have been solved: (i) As our approach is based upon TGGs, consistent specifications of bidirectional graph transformations are guaranteed. (ii) Our approach provably complies to important formal properties and (iii) scales polynomially with the number of changes and affected elements and, therefore, is efficient. (iv) The incremental algorithm determines the set of changes and their dependent elements and, therefore, preserves user added information in unaffected elements. (v) As our algorithm supports non-functional sets of TGG rules and allows for complex rules, it can be considered as expressive.

From a practical point of view, the approach could be employed in a real-world scenario such as described in Chap. 2. Here, we collected numerous requirements during our research cooperation with Siemens in the domain of automation engineering. A fully-fledged transformation approach should (i) be applicable in either direction (bidirectional), (ii) perform efficiently, (iii) be traceable regarding its results, (iv) be specified declaratively, (v) be applicable in batch mode and (vi) incrementally while preserving information, (vii) support concurrent model modifications, (viii) be validated and (ix) comply with formal properties. What remains open for future work is to enable concurrent model modifications. As this requires to detect and to cope with contradictions between individual changes, this was out-of-scope for this thesis.

EVALUATION OF THE SOLUTION

Regarding current TGG approaches, the motivated requirements are already met by some other implementations. Formal properties are tackled by the research group of Hartmut Ehrig [EEHP09], efficiency is a main feature of the implementation of the working group of Holger Giese [GH09] and preserving user attached information is achieved by the research group of Wilhelm Schäfer [GPR11]. Unfortunately, none of these approaches combines all desired requirements in one implementation. Especially extending TGG theory with additional features and prove that all formal properties are still met, is a hard task to fulfill.

For these reasons, we decided to develop a new TGG dialect, which is efficient and information preserving but can also be systematically verified for its compliance to formal properties. The challenge behind efficiency and information preserving capabilities is to identify an appropriate processing order for the applied changes and their affected elements. Hence, model elements have to be ordered to avoid multiple re-synchronization runs for the same elements for efficiency reasons.

In this thesis, a novel TGG approach was developed which is capable of solving the described challenges: The basic idea is to analyze the declarative TGG specification and to collect relevant information from specific patterns in the TGG rules. Such information can further be used at runtime to define a partial order of a given model that shall be synchronized with an opposite model. As this partial order denotes the dependency relationships between the model elements and, therefore, allows for determining which elements shall precede others in the transformation sequence, we named this approach *precedence-driven TGGs*.

Furthermore, this partial order enables the incremental control algorithm to determine an appropriate traversal order. The (batch and incremental) algorithms presented in this thesis fulfill this task provably in polynomial runtime while retaining all formal properties. Although an additional model difference recognition has to be applied in advance of a synchronization, we were able to argue that this does not significantly affect the runtime complexity. In contrast, relying for example on traces of model changes, unnecessary and complementary synchronization actions may be applied that may introduce additional overhead to the transformation process. In addition, the runtime complexity depends on the number of elements that have to be processed during a single synchronization run. The runtime complexity of the incremental algorithm is, therefore, only influenced by the number of changed and their dependent elements. We have been able to show that our approach determines a considerable superset of dependent elements which provably guarantees that all sufficient elements are processed while trying to reduce this number as close as possible to the number of actual necessary elements. Hence, the formal properties guarantee that the algorithm produces only correct models (or terminates with an appropriate exception) and is able to handle all models as input which could have been created by the algorithm (i.e., completeness).

A major goal was to preserve user information while synchronizing models. As we believe that TGGs should have a certain degree of freedom regarding the produced models, we do not require functional behavior which forces TGG developers to build specifications that always produce the same output considering an input model. In contrast, we allow users to specify TGGs that are capable of produc-

ing a set of correct output models. While processing the input model, the user has to either provide certain heuristics that the control algorithm is able to decide which specific rule has to be applied, or the user has to decide himself on-demand. In both cases, certain decisions have been made that have to be retained while a synchronization is performed. This has been achieved with precedence TGGs as well.

FUTURE WORK

Nevertheless, the approach presented in this thesis has some deficiencies that have to be tackled in future contributions to complete this approach: A minor open issue is the integration of edges as first-class objects. Currently, only such rules are allowed in combination with our approach that create an edge together with an adjacent node. Another restriction of expressiveness is that Negative Application Conditions (NACs) for describing global constraints are not yet supported. First thoughts in this direction showed that introducing NACs can be achieved in a straight-forward manner as such NACs do not influence the ordering of the input model. Nevertheless, we have to prove that the use of NACs still guarantees all formal properties also in the case of incremental model synchronization.

Another current drawback is that only such model changes are supported that consist of additions and deletions only (i.e., element modifications are treated as a delete old and add new element sequence). In reality, this is insufficient as elements could also be adjusted or moved within the model. Such changes should also be efficiently handled by our TGG implementation. Therefore, further operational rules have to be derived which, for example, propagate an attribute change to the opposite domain and apply such actions at the right point in the synchronization control algorithm.

Finally, concurrent changes to models is another open issue. The interesting aspect of this challenge is that indirectly each other contradicting changes may influence each other and sophisticated propagation strategies have to guarantee that the updated models are consistent to each other.

Part VII

APPENDIX

BIBLIOGRAPHY

[AKK+08] Carsten Amelunxen, Felix Klar, Alexander Königs, Tobias Rötschke, and Andy Schürr. Metamodel-Based tool Integration with MOFLON. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 807–810, New York, NY, USA, 2008. ACM.

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Arend Rensink and Jos Warmer, editors, *Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA '06)*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Berlin / Heidelberg, Germany, 2006. Springer.

[ALPS11] Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. eMoflon : Leveraging EMF and Professional CASE Tools. In *3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe '11) in Proceedings of Informatik 2011*, volume 192 of *Lecture Notes in Informatics (LNI)*, page 281, Bonn, Germany, 2011. Gesellschaft für Informatik e.V.

[ALS12] Anthony Anjorin, Marius Lauder, and Andy Schürr. eMoflon: A Metamodelling and Model Transformation Tool. In Harald Störrle, Goetz Botterweck, Michel Bourdellès, Dimitrios S. Kolovos, Richard F. Paige, Ella Roubtsova, Julia Rubin, and Juha-Pekka Tolvanen, editors, *Joint Proceedings of the Co-located Events at the 8th European Conference on Modelling Foundations and Applications (ECMFA '12)*, page 348, Copenhagen, Denmark, 2012. Technical University of Denmark (DTU).

[AST12] Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. Construction of Integrity Preserving Triple Graph Grammars. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proceedings of the 6th International Conference on Graph Transformation (ICGT '12)*, volume 7562 of *Lecture Notes in Computer Science (LNCS)*, pages 356–370, Berlin / Heidelberg, Germany, 2012. Springer.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1986.

[AVS12] Anthony Anjorin, Gergely Varró, and Andy Schürr. Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. In Frank Hermann and Janis Voigtländer, editors, *First International Workshop on Bidirectional Transformations (bx '12)*, volume 49 of *Electronic Communications of the EASST (ECEASST)*, pages 1–16. EASST, 2012.

[Baloo] Helmut Balzert. *Lehrbuch der Software-Technik: Software Entwicklung*. Spektrum Akademischer Verlag, Heidelberg / Berlin, Germany, 2nd edition, 2000.

[BdBfI10] Die Beauftragte der Bundesregierung für Informationstechnik. V-Modell XT Bund 1.0, 2010. Bundesverwaltungsamt, Bundesstelle für Informationstechnik.

[BET08] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS '08)*, volume 5301 of *Lecture Notes in Computer Science (LNCS)*, pages 53–67, Berlin / Heidelberg, Germany, 2008. Springer.

[BFP+08] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang : Resourceful Lenses for String Data. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*, volume 43, pages 407–419, New York, NY, USA, 2008. ACM.

[BP08] Cédric Brun and Alfonso Pierantonio. Model Differences in the Eclipse Modelling Framework. *UPGRADE – European Journal for the Informatics Professional*, IX(2):29–34, 2008.

[BTS00] Paolo Bottoni, Gabriele Taentzer, and Andy Schürr. Efficient Parsing of Visual Languages Based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proceedings of the IEEE International Symposium on Visual Languages 2000 (VL '00)*, pages 59–60, Washington, DC, USA, 2000. IEEE Computer Society.

[BWW11] Thomas Buchmann, Bernhard Westfechtel, and Sabine Winetzhammer. The Added Value of Programmed Graph Transformations - A Case Study From Software Configuration Management. In Andy Schürr, Daniel Varró, and Gergely Varró, editors, *Proceedings of the 4th International Symposium on Applications and Graph Transformations with Industrial Relevance (AGTIVE '11)*, volume 7233 of *Lecture Notes in Computer Science (LNCS)*, pages 198–209, Berlin / Heidelberg, Germany, 2011. Springer.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal - Model-Driven Software Development*, 45(3):621–645, 2006.

[dLV02a] Juan de Lara and Hans Vangheluwe. AToM3: A Tool for Multi-Formalism and Meta-Modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE '05)*, volume 2306 of *Lecture Notes in Computer Science (LNCS)*, pages 174–188, Berlin / Heidelberg, Germany, 2002. Springer.

[dLV02b] Juan de Lara and Hans Vangheluwe. Using AToM3 as a Meta-Case Tool. In *Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS '02)*, pages 642–649, 2002.

[DV07] Marcos Didonet Del Fabro and Patrick Valduriez. Semi-Automatic Model Integration Using Matching Transformations and Weaving Models. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC '07)*, pages 963–970, New York, NY, USA, 2007. ACM.

[DXC+11] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS '11)*, volume 6981 of *Lecture Notes in Computer Science (LNCS)*, pages 304–318. Springer, Berlin / Heidelberg, Germany, 2011.

[EEE+07] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information Preserving Bidirectional Model Transformations. In Matthew B. Dwyer and Antónia Lopes, editors, *Proceedings of the 10th*

International Conference on Fundamental Approaches to Software Engineering (FASE '07), volume 4422 of *Lecture Notes in Computer Science (LNCS)*, pages 72–86, Berlin / Heidelberg, Germany, 2007. Springer.

[EEH08] Hartmut Ehrig, Karsten Ehrig, and Frank Hermann. From Model Transformation to Model Integration Based on the Algebraic Approach to Triple Graph Grammars. In Claudia Ermel, Juan de Lara, and Reiko Heckel, editors, *Proceedings 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT '08)*, volume 10 of *Electronic Communications of the EASST (ECEASST)*, pages 1–14. EASST, 2008.

[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, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*, volume 5795 of *Lecture Notes in Computer Science (LNCS)*, pages 241–255, Berlin / Heidelberg, Germany, 2009. Springer.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, Berlin / Heidelberg, Germany, 1st edition, 2006.

[EGLT11] Claudia Ermel, Jürgen Gall, Leen Lambers, and Gabriele Taentzer. Modeling with Plausibility Checking: Inspecting Favorable and Critical Signs for Consistency Between Control Flow and Functional Behavior. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering (FASE '11)*, volume 6603 of *Lecture Notes in Computer Science (LNCS)*, pages 156–170, Berlin / Heidelberg, Germany, 2011. Springer.

[EHGB12] Claudia Ermel, Frank Hermann, Jürgen Gall, and Daniel Binanzer. Visual Modeling and Analysis of EMF Model Transformations based on Triple Graph Grammars. In *Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs '12) (accepted for publication)*, Electronic Communications of the EASST (ECEASST), pages 1–12. EASST, 2012.

[FF03] John Fuegi and Jo Francis. Lovelace & Babbage and the Creation of the 1843 'Notes'. *IEEE Annals of the History of Computing*, 25(4):16–26, 2003.

[FGM+05] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View Update Problem. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, 40(1):233–246, 2005.

[FHKS09] Jan Friedrich, Ulrike Hammerschall, Marco Kuhrmann, and Marco Sihling. *Das V-Modell XT*. Informatik im Fokus. Springer, Berlin / Heidelberg, Germany, 2nd edition, 2009.

[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 Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformations (TAGT '98)*, volume 1764 of *Lecture Notes on Computer Science (LNCS)*, pages 296–309, Berlin / Heidelberg, Germany, 2000. Springer.

[FPP08] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient Lenses. *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, 43(9):383–396, 2008.

[Fra76] Reinhold Franck. PLAN2D - Syntactic Analysis of Precedence Graph Grammars. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages (POPL '76)*, pages 134–139, New York, NY, USA, 1976. ACM.

[Fra78] Reinhold Franck. A Class of Linearly Parsable Graph Grammars. *Acta Informatica*, 10(2):175–201, 1978.

[FW07] Sabrina Förtsch and Bernhard Westfechtel. Differencing and Merging of Software Diagrams State of the Art and Challenges. In Joaquim Filipe, Boris Shishkov, and Markus Helfert, editors, *Proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT '07)*, pages 90–99, Setubal, Portugal, 2007. INSTICC Press.

[GdL04] Esther Guerra and Juan de Lara. Event-Driven Grammars: Towards the Integration of Meta-modelling and

Graph Transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Proceedings of the 2nd International Conference on Graph Transformations (ICGT '04)*, volume 3256 of *Lecture Notes in Computer Science (LNCS)*, pages 54–69, Berlin / Heidelberg, Germany, 2004. Springer.

[GEH10] Ulrike Golas, Hartmut Ehrig, and Frank Hermann. Enhancing the Expressiveness of Formal Specifications for Model Transformations by Triple Graph Grammars with Application Conditions. In Rachid Echahed, Annegret Habel, and Mohamed Mosbah, editors, *Proceedings of the 3rd International Workshop on Graph Computation Models (GCM' 10)*, pages 149–164, 2010.

[GH09] Holger Giese and Stephan Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical Report 28, Hasso-Plattner-Institut, Potsdam, Germany, 2009.

[GHL10] Holger Giese, Stephan Hildebrandt, and Leen Lambers. Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. In *Proceedings of the 2010 Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa '10)*, pages 19–24, Washington, DC, USA, 2010. IEEE Computer Society.

[GK07] Rubino Geiss and Moritz Kroll. On Improvements of the Varró Benchmark for Graph Transformation Tools. Technical Report 2007-07, Universität Karlsruhe, Karlsruhe, Germany, 2007.

[GPR11] Joel Greenyer, Sebastian Pook, and Jan Rieke. Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In Robert France, Jochen M. Kuester, Behzad Bordbar, and Richard F. Paige, editors, *Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA '11)*, volume 6698 of *Lecture Notes in Computer Science (LNCS)*, pages 144–159, Berlin / Heidelberg, Germany, 2011. Springer.

[GR11] Joel Greenyer and Jan Rieke. An Improved Algorithm for Preventing Information Loss in Incremental Model Synchronization. Technical Report tr-ri-11-324, University of Paderborn, Paderborn, Germany, 2011.

[Gro09] Richard C. Gronback. *eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Longman Publishing Co., Inc., Amsterdam, The Netherlands, 1st edition, 2009.

[GSR05] Leif Geiger, Christian Schneider, and Carsten Reckord. Template- and Modelbased Code Generation for MDA-Tools. In Holger Giese and Albert Zündorf, editors, *Proceedings of the 3rd International Fujaba Days 2005*, pages 57–62, 2005.

[Gua09] Yuan Guan. Vergleich von QVT-Implementierungen mit TGG. Master's thesis, Technische Universität Darmstadt, Germany, 2009.

[GW06] Holger Giese and Robert Wagner. Incremental Model Synchronization with Triple Graph Grammars. In Oscar Nierstrasz, John Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS '09)*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 543–557, Berlin / Heidelberg, Germany, 2006. Springer.

[GW08] Holger Giese and Robert Wagner. From Model Transformation to Incremental Bidirectional Model Synchronization. *Software & Systems Modeling*, 8(1):21–43, 2008.

[HBJ09] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. COPE - Automating Coupled Evolution of Metamodels and Models. In Sophia Drossopoulou, editor, *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP '09)*, volume 5653, pages 52–76, Berlin / Heidelberg, Germany, 2009. Springer.

[HEEO11] Frank Hermann, Hartmut Ehrig, Claudia Ermel, and Fernando Orejas. Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars - Extended Version. Technical Report 2011-14, Technische Universität Berlin, Germany, Berlin, 2011.

[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 Jean Bézivin, Mark Richard Soley, and Antonio Vallecillo, editors, *Proceedings of the 1st International Workshop on Model-Driven Interoperability (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 Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Proceedings of the 14th International Conference on Model Driven Engineering Languages*

and Systems (MODELS '11), volume 6981 of *Lecture Notes in Computer Science (LNCS)*, pages 668–682, Berlin / Heidelberg, Germany, 2011. Springer.

[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 5th International Conference on Graph Transformations (ICGT '10)*, volume 6372 of *Lecture Notes in Computer Science (LNCS)*, pages 155–170, Berlin / Heidelberg, Germany, 2010. Springer.

[Her11] Markus Herrmannsdoerfer. COPE - A Workbench for the Coupled Evolution of Metamodels and Models. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Proceedings of the 3rd International Conference on Software Language Engineering (SLE '10)*, volume 6563 of *Lecture Notes in Computer Science (LNCS)*, pages 286–295, Berlin / Heidelberg, Germany, 2011. Springer.

[HH08] Reinhard Höhn and Stephan Höppner. *Das V-Modell XT*. eXamen.press, Berlin / Heidelberg, Germany, 1st edition, 2008.

[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 '10)*, volume 45 of *ICFP '10*, pages 205–216, New York, NY, USA, 2010. ACM.

[HHI+11a] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. GRoundTram Version 0.9.2 User Manual. 2011.

[HHI+11b] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. In Perry Alexander, Corina Pasareanu, and John Hosking, editors, *Proceedings of the 26th International Conference on Automated Software Engineering (ASE '11)*, pages 480–483, Washington, DC, USA, 2011. IEEE Computer Society.

[HHKN09] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. A Compositional Approach to Bidirectional Model Transformation. In *Companion Volume*

of the 31st International Conference on Software Engineering (ICSE '09), pages 235–238, Washington, DC, USA, 2009. IEEE Computer Society.

[HLG+12] Stephan Hildebrandt, Leen Lambers, Holger Giese, Dominic Petrick, and Ingo Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In *Proceedings of the 4th International Symposium on Applications and Graph Transformations with Industrial Relevance (AGTIVE '11)*, volume 7233 of *Lecture Notes in Computer Science (LNCS)*, pages 238–253, Berlin / Heidelberg, Germany, 2012. Springer.

[Hof11] Matthias Hof. *Grafische Modellierung semantischer Abbildungen zwischen Datenmodellen unterschiedlicher PDM-Systeme und automatische Ableitung von Konvertierungsroutinen*. Bachelor's thesis, Technische Universität Darmstadt, Germany, 2011.

[HSST11] Zhenjiang Hu, Andy Schürr, Perdita Stevens, and James Terwilliger. Bidirectional Transformations "bx" (Dagstuhl Seminar 11031). *Dagstuhl Reports*, 1(1):42–67, 2011.

[IEC96] IEC. Industrial Systems, Installations and Equipment and Industrial Products. Structuring Principles and Reference Designations - Part 1: Basic Rules (IEC 61346-1), 1996. International Electrotechnical Commission (IEC).

[IEC03] IEC. Programmable Controllers - Part 3: Programming Languages 2003 (IEC 61131-3), 2003. International Electrotechnical Commission (IEC).

[IEC09] IEC. Industrial Systems, Installations and Equipment and Industrial Products. Structuring Principles and Reference Designations - Part 1: Basic Rules (IEC 81346-1), 2009. International Electrotechnical Commission (IEC).

[ikv12] ikv++ technologies ag. mediniQVT, 2012. `http://projects.ikv.de/qvt`, last accessed: November 9th, 2012.

[ISO86] ISO. ISO 8879:1986 Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML), 1986. International Organization for Standardization (ISO).

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming: Special Issue on Second Issue of Experimental Software and Toolkits (EST)*, 72(1-2):31–39, 2008.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Revised Selected Papers of the Satellite Events at the MODELS 2005 Conference (MODELS '05)*, volume 3844 of *Lecture Notes in Computer Science (LNCS)*, pages 128–138, Berlin / Heidelberg, Germany, 2006. Springer.

[Kö5] Alexander Königs. Model Transformation with Triple Graph Grammars. In *Proceedings of the International Workshop on Model Transformations in Practice (MTIP '05)*, 2005.

[Kau83] Manfred Kaul. Parsing of Graphs in Linear Time. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the 2nd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science (LNCS)*, pages 206–218, Berlin / Heidelberg, Germany, 1983. Springer.

[Kau86] Manfred Kaul. *Syntaxanalyse von Graphen bei Präzedenz-Graph-Grammatiken*. Dissertation thesis, Osnabrück, Germany, 1986.

[Kau87] Manfred Kaul. Practical Applications of Precedence Graph Grammars. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science (LNCS)*, pages 326–342, Berlin / Heidelberg, Germany, 1987. Springer.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07)*, pages 285–294, New York, NY, USA, 2007. ACM.

[KL10] Carsten Kestermann and Timo Leimbach. Software-Monitor Deutschland 2010, 2010. Fraunhofer ISI.

[Kla12] Felix Klar. *Efficient and Compatible Bidirectional Formal Language Translators based on Extended Triple Graph Grammars*. Dissertation thesis, Technische Universität Darmstadt, Germany, 2012.

[KLKS10] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In Andy Schürr,

Claus Lewerentz, Gregor Engels, Wilhelm Schäfer, and Bernhard Westfechtel, editors, *Graph Transformations and Model Driven Enginering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *Lecture Notes in Computer Science*, pages 141–174. Springer, Berlin / Heidelberg, Germany, November 2010.

[KLS+12] Tina Krausser, Marius Lauder, Michael Schlereth, Ulrich Epple, and Andy Schürr. Integrated Graph Transformations in Automation Systems. In Inge Troch, editor, *Proceedings of 7th International Conference on Mathematical Modelling (MATHMOD '12)*, Vienna, Austria, 2012. IFAC.

[KNNZ99] Thomas Klein, Ulrich Nickel, Jörg Niere, and Albert Zündorf. From UML to Java And Back Again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, 1999.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management (GaMMa '06)*, pages 13–20, New York, NY, USA, 2006. ACM.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT '08)*, volume 5063 of *Lecture Notes in Computer Science (LNCS)*, pages 46–60, Berlin / Heidelberg, Germany, 2008. Springer.

[KRPGD11] Dimitrios S. Kolovos, Louis Rose, Richard F. Paige, and Antonio García-Domínguez. *The Epsilon Book*. The Eclipse Foundation, 2011.

[KRS09] Felix Klar, Sebastian Rose, and Andy Schürr. TiE - A Tool Integration Environment. In *Proceedings of the 5th ECMDA Traceability Workshop (ECMDA-TW '09)*, volume WP09-09 of *CTIT Workshop Proceedings*, pages 39–48, Enschede, The Netherlands, 2009. University of Twente.

[KRW04] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. An Adaptable TGG Interpreter for In-Memory Model Transformations. In Andy Schürr and Albert Zündorf, editors, *Proceedings of the 2nd International Fujaba Days*, pages 35–38, Paderborn, Germany, 2004. University of Paderborn, University of Paderborn.

[LAVS12a] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Bidirectional Model Transformation with Precedence Triple Graph Grammars. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos, editors, *Proceedings of the 8th International Conference on Modelling Foundations and Applications (ECMFA '12)*, volume 7349 of *Lecture Notes in Computer Science (LNCS)*, pages 287–302, Berlin / Heidelberg, Germany, 2012. Springer.

[LAVS12b] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Efficient Model Synchronization with Precedence Triple Graph Grammars. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proceedings of the 6th International Conference on Graph Transformation (ICGT '12)*, volume 7562 of *Lecture Notes in Computer Science (LNCS)*, pages 401–415, Berlin / Heidelberg, Germany, 2012. Springer.

[Lei10] Timo Leimbach. Software-Atlas Deutschland 2010, 2010. Fraunhofer ISI.

[LL06] Jochen Ludewig and Horst Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, Heidelberg, Germany, 1st edition, 2006.

[LSRS10] Marius Lauder, Michael Schlereth, Sebastian Rose, and Andy Schürr. Model-Driven Systems Engineering: State-of-the-Art and Research Challenges. *Bulletin of the Polish Academy of Sciences, Technical Sciences*, 58(3):409–422, 2010.

[LW11] Timo Leimbach and Sven Wydra. Software-Atlas Deutschland 2011, 2011. Fraunhofer ISI.

[MKR06] Tom Mens, Günter Kniesel, and Olga Runge. Transformation Dependency Analysis - A Comparison of Two Approaches. *Série L'objet- logiciel, base de données, réseaux*, 12(HS):167–182, 2006.

[MM03] Jishnu Miller and Joaquin Mukerji. MDA Guide Version 1.0.1, 2003. Object Management Group (OMG).

[MvdSD06] Tom Mens, Ragnhild van der Straeten, and Maja D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*

(MODELS '06), volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 200–214, Berlin / Heidelberg, Germany, 2006. Springer.

[Nag79a] Manfred Nagl. A Tutorial and Bibliographical Survey on Graph Grammars. In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *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 70–126, Berlin / Heidelberg, Germany, 1979. Springer.

[Nag79b] Manfred Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, Wiesbaden, Germany, 1979.

[Nie94] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, Burlington, MA, USA, 11th edition, 1994.

[OMG06] OMG. Meta Object Facility (MOF) Core Specification 2.0, 2006. Object Management Group (OMG).

[OMG08] OMG. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification, 2008. Object Management Group (OMG).

[OMG10] OMG. MOF Facility Object Lifecycle (MOFFOL), 2010. Object Management Group (OMG).

[OMG11] OMG. Unified Modeling Language (UML), v2.4.1, 2011. Object Management Group (OMG).

[OMG12] OMG. OMG Systems Modeling Language (SysML), v1.3, 2012. Object Management Group (OMG).

[Pra71] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 5(6):560–595, 1971.

[RLSS11] Sebastian Rose, Marius Lauder, Michael Schlereth, and Andy Schürr. A Multidimensional Approach for Concurrent Model Driven Automation Engineering. In Janis Osis and Erika Asnina, editors, *Model-Driven Domain Analysis and Software Development: Architectures and Functions*, pages 90–113. IGI Global, Hershey, PA, USA, 2011.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Longman Publishing Co., Inc., Amsterdam, The Netherlands, 2nd edition, 2009.

[Sch95] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '94)*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163, Berlin / Heidelberg, Germany, 1995. Springer.

[SG08] Maik Schmidt and Tilman Gloetzner. Constructing Difference Tools for Models Using the SiDiff Framework. In *Companion of the 30th International Conference on Software Eengineering (ICSE '08)*, pages 947–948, New York, NY, USA, 2008. ACM.

[Sie10] Siemens Industry Software GmbH & Co.KG. SIMATIC STEP7 Programming Software, 2010. `http://www.automation.siemens.com/simatic/industriesoftware/html_76/products/step7.htm`, last accessed: July 19th, 2010.

[Sie12] Siemens Industry Software GmbH & Co.KG. Comos Software Solutions, 2012. `http://www.automation.siemens.com/mcms/plant-engineering-software/en/Pages/Default.aspx?&L=1`, last accessed: October 30th, 2012.

[SK08] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Proceedings of the 4th International Conference on Graph Transformation (ICGT '08)*, volume 5214 of *Lecture Notes in Computer Science (LNCS)*, pages 411–425, Berlin / Heidelberg, Germany, 2008. Springer.

[SK12] Michael Schlereth and Tina Krausser. Platform-Independent Specification of Model Transformations @ Runtime Using Higher-Order Transformations. In Elmar J. Sinz and Andy Schürr, editors, *Proceedings of the National Workshop Modellierung 2012*, volume 201 of *Lecture Notes in Informatics (LNI)*, pages 139–154, Bonn, Germany, 2012. Gesellschaft für Informatik e.V.

[SLRS10] Michael Schlereth, Marius Lauder, Sebastian Rose, and Andy Schürr. Concurrent Model Driven Automation Engineering - Building Engineering Tool Integration Systems. *atp edition - Automatisierungstechnische Praxis*, 52(11):64–69, 2010.

[Som07] Ian Sommerville. *Software Engineering*. Pearson Education, Upper Saddle River, NJ, USA, 8th edition, 2007.

[SR09] Andrew P. Sage and William B. Rouse. *Handbook of Systems Engineering and Management*. John Wiley & Sons, San Francisco, CA, USA, 2nd edition, 2009.

[SRS09] Michael Schlereth, Sebastian Rose, and Andy Schürr. Model Driven Automation Engineering - Characteristics and Challenges. In Holger Giese, Michaela Huhn, Ulrich Nickel, and Bernhard Schätz, editors, *Proceedings of the Dagstuhl-Workshop Modellbasierte Entwicklung eingebetteter Systeme (MBEES '09)*, volume 2009-01 of *Informatik Berichte*, pages 1–15, Braunschweig, Germany, 2009. Technische Universität Braunschweig.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Berlin / Heidelberg, Germany, 1st edition, 1973.

[Ste08a] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In Ralf Lämmel, Joost Visser, and Joao Saraiva, editors, *Revised Papers of the 2nd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE '07)*, volume 5235 of *Lecture Notes in Computer Science (LNCS)*, pages 408–424, Berlin / Heidelberg, Germany, 2008. Springer.

[Ste08b] Perdita Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software & Systems Modeling*, 9(1):7–20, 2008.

[Ste08c] Perdita Stevens. Towards an Algebraic Theory of Bidirectional Transformations. In *Proceedings of the 4th International Conference on Graph Transformations (ICGT '08)*, volume 5214 of *Lecture Notes in Comuter Science (LNCS)*, pages 1–17, Berlin / Heidelberg, Germany, 2008. Springer.

[TEG+05] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Daniel Varró, and Szilvia Varró-Gyapay. Model Transformations by Graph Transformations: A Comparative Study. In *Proceedings of the International Workshop on Model Transformations in Practice (MTIP '05)*, pages 1–48, 2005.

[TV11] Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *The Journal of Object Technology*, 10(5):1–29, 2011.

[Ulm09] Renate Ulm, editor. *Die 9 Symphonien Beethovens*. Bärenreiter, Kassel, Germany, 6th edition, 2009.

[VDI04a] VDI. VDI 2206: Design Methodology for Mechatronic Systems, 2004. Verband Deutscher Ingenieure (VDI).

[VDI04b] VDI. VDI 4499: Digital Factory - Fundamentals, 2004. Verband Deutscher Ingenieure (VDI).

[vDRH+11] Markus von Detten, Jan Rieke, Christian Heinzemann, Dietrich Travkin, and Marius Lauder. A new Meta-Model for Story Diagrams. In Ulrich Norbisrath and Ruben Jubeh, editors, *Proceedings of the 8th International Fujaba Days*, Kasseler Informatikschriften (KIS), pages 1–15, Kassel, Germany, 2011. Universität Kassel.

[VSC06] Markus Voelter, Thomas Stahl, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, San Francisco, CA, USA, 2006.

[VSV05] Gergely Varró, Andy Schürr, and Daniel Varró. Benchmarking for Graph Transformation. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC '05)*, pages 79–88, Washington, DC, USA, 2005. IEEE Computer Society.

[WDC03] Yuan Wang, David J. Dewitt, and Jin-Yi Cai. X-Diff : An Effective Change Detection Algorithm for XML Documents. In *Proceedings of the 19th International Conference on Data Engineering*, pages 519–530, Washington, DC, USA, 2003. IEEE Computer Society.

[Wei06] Tim Weilkins. *Systems Engineering mit SysML/UML*. dpunkt.verlag, Heidelberg, Germany, 1st edition, 2006.

[Wes10] Bernhard Westfechtel. A Formal Approach to Three-Way Merging of EMF Models. In *Proceedings of the 1st International Workshop on Model Comparison in Practice (IWMCP '10)*, pages 31–41, New York, NY, USA, 2010. ACM.

[Wir08] Niklaus Wirth. A Brief History of Software Engineering. *IEEE Annals of the History of Computing*, 30(3):32–39, 2008.

[WKK+11] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitrios S. Kolovos, Richard F. Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In Jordi Cabot and Eelco Visser,

editors, *Proceedings of the 4th International Conference on Theory and Practice of Model Transformations (ICMT '11)*, volume 6707 of *Lecture Notes in Computer Science (LNCS)*, pages 31–46, Berlin / Heidelberg, Germany, 2011. Springer.

[WKK+12] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitrios S. Kolovos, Richard F. Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. Surveying Rule Inheritance in Model-to-Model Transformation Languages. *Journal of Object Technology*, 11(2):1–46, 2012.

[WW66] Niklaus Wirth and Helmut Weber. EULER: A Generalization of ALGOL and its Formal Definition: Part 1. *Communications of the ACM*, 9(1):13–25, 1966.

[XS05] Zhenchang Xing and Eleni Stroulia. UMLDiff. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pages 54–65, New York, NY, USA, 2005. ACM.

CURRICULUM VITAE

MARIUS LAUDER
born on June, 17th 1982 in Halle/Saale
email: marius.lauder@es.tu-darmstadt.de

2008 - 2013	Doctoral researcher and fellowship holder of the Graduate School of Computational Engineering of the Technische Universität Darmstadt.
2006 - 2008	M.Sc. in Computer Science from the Technische Universität Darmstadt. Thesis topic: “Extracting Domain-Specific Thesauri from Wikipedia and Wiktionary"
2003 - 2006	B.Sc. in Computer Science from the Johannes Gutenberg-Universität Mainz. Thesis topic: “Generierung von vorlesbarem Text aus LaTeX-Quelltext” (Building a compiler for LaTeX source code to machine readable text for visually impaired people)
1993 - 2002	A-levels from the private Johannes Gymnasium Lahnstein