Concept and Transformationen of an Application Environment for Model Transformations Based on Triple Graph Grammars and Mathematica.

University: Technische Universität Berlin

Institute: Institut für Softwaretechnik und Theoretische Informatik

Author: Olegs Klujs

Advisers: Hartmut Ehrig, Claudia Ermel

Date/Place: 17.08.2010 / Berlin

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt

Berlin, den

….....

Unterschrift

Abstract

This thesis provides a description of an application environment realisation for model transformation / integration based on the triple graph approach. The implementation is performed in Mathematica. Due to the use of functional programming in Mathematica, the resulting application shows a low abstraction from the theoretical concepts. The implementation is tested in two case studies for model transformation between class and entity-relationship diagrams. Additionally, the basis for a third case study is introduced. This case study will apply the model transformation and integration to business service and IT service models by the use of ABT-Reo diagrams.

Kurzbeschreibung

In dieser Diplomarbeit wird die Realisierung einer Anwendungsumgebung für Modelltransformationen bei Verwendung des Konzeptes der Tripel Graphen beschrieben. Die Applikation zeichnet eine niedrige Abstraktion von den theoretischen Grundlagen aus. Die Möglichkeit einer solchen Implementierung ist auf das funktionale Programmieren mit Mathematica zurückzuführen. Die Implementierung wurde in zwei Fallbeispielen getestet. In den Fallbeispielen wird die Transformation von Klassen- in Entity-Relationship Diagramme behandelt. Zusätzlich wurde die Grundlage für ein drittes Fallbeispiel zur Realisierung von Modelltransformationen zwischen Geschäfts- und IT Service Modellen unter Verwendung von ABT-Reo Diagrammen gelegt.

CONTENTS

I. INTRODUCTION

1 Motivation

Nowadays visual modelling is an irreplaceable part of any business process, technical design and production, system analyses and modification. No business, no company, no project can be started without an analysis of work flows and a categorisation of units. Due to the visual modelling proved to be more observable and distanced – a proper generalised technique for working with huge data flows, relationships between units, simulation of process execution, etc. - there have been developed various visual modelling languages and techniques to realize different constructs and models in different systems and business branches.

The understanding of the need and efficiency of the visual modelling led to a wide spread implementation of such techniques and fast growth in the complexity of developed systems. At this point a major problem came to surface: a system can be designed and implemented as good as possible, however in case there is no suitable interface to the environment of the system (other systems) – it stays inefficient.

This led to the development of UML with implementations like RUP (Rational Unified Process) and Fusion, or DSL with various software realisations for designing business / development processes and interaction (Visio, SmartDraw, etc.), which aims to unite different modelling techniques under one source. In UML it is the language which shall provide the interface, in DSL interfaces appear during the language generation process.

However UML and DSL loose their efficiency by model growth or deeper specification in some business areas. UML provides very abstract/general techniques, when the developers may seek for a more detailed / less abstract specification for their Diagrams. DSL reaches its borders, when projects grow and the interfaces between different business branches become less obvious, or a present realisation has to be extended / reorganised. This fact creates a slot for a new technique.

Generally the requirement can be defined as follows: a technique is needed, which provides a possibility to create interfaces between visual Model of different kinds and languages. Such an approach seek the model integration / transformation method based on algebraic approach to Triple Graph Grammars as described in [EEH+08]. The main idea of this method is that any visual model can be converted into a graph, which is build up according to the rules of some formal language. Furthermore, any such graph can be interpreted as a unique sequence of basic (atomic) rules. These rules provide possibility (after the basic rules for a visual model are defined) to search for a correspondence between rules of different languages. Having corresponding atomic rules and following the build up sequence of a given graph in one visual language, we build up an appropriate graph in the corresponding language. In such a way a “bridge” between two model of two different visual languages can be constructed.

2 Overview

The thesis consist of five parts.

The first part is the introduction, which includes a motivation part and an the overview of the thesis.

In the second part the theoretical background of graphs, triple rules and triple graph grammars as introduced in [Sch94] and [EEH+08] are reviewed, as well as techniques of model transformation and integration. In the end of the second part, in chapter II. 1.5. , the triple rule concept is extended by a DPO rule concept, which extends triple rules with deletion the functionality by keeping the morphism inductivity.

In the third part, a short introduction into Mathematica methods and concepts, used during the implementation is made. In the next section the implementation of attributed (typed) graph transformation in Mathematica realised by Jochen Adamek is briefly introduced and the methods, included in the implementation in this thesis are described. Section III. 3 is dedicated to the realization of model transformation / integration. The implementation of all theoretical concepts from the second part are explained step by step Section three also contains the implementation of the DPO rule extension, a method form automated match search and automated forward / integration rule sequence application.

The implemented concepts are tested in two case studies in part four. In these case studies, two examples of class to entity-relationship diagrams transformation / integration execution are described. It is shown how the rules are build, how the models are generated, how automated source (source-/target-) search are applied to the model and how the automated forward and integration sequences are applied.

In the fifth part the future works can be found. The approach of ABT-Reo case study realisation is briefly described.

II. REVIEW OF THEORETICAL BACKGROUND

1 Triple Graph Grammars

Working with visual information representation has definite advantages, compared to dealing with written information. This advantages led to development of various visual languages in all areas of modern science. Only in informatics there are countless kinds of diagrams, used to represent data flows, data storage, user interactions, instruction sequences, operation system levels, system restrictions, etc. Yet, despite of all opportunities we gain by using State-Charts, Sequence-, Entity-/Relationship-, Class- Diagrams, Fraction charts or Data Graphs, etc. we are limited by the syntax of chosen visualisation method.

Triple Graph Grammars approach (first introduced by Andy Schürr[Sch94]) is an attempt to create a method for connecting different systems/models with respect to some predefined rules/criterias, so that changes in one system/model would inevitably lead to changes in the other. The general idea of this approach is to define a three-tuple object out of two graphs, a connections-relationship between them (realised also as a graph) and two morphisms which carry out the transformation of one graph into an other inside the object. On the one hand, this three-tuple can possess its own specific characteristic and behaviour type; on the other hand graphs inside the tuple stay independent from each other since the connection between them is realized with a third graph and morphisms. Thus two diagrams from different visualisation languages can be connected, compared or put in dependence of each other by the modification methods staying specific for chosen visualisation languages.

This chapter is a short introduction to the theoretical background of Triple Graph Grammars which have been realized or used to realize Triple Graph Transformation/Integration in “Mathematica” in this thesis.

First formal definitions of ***Triple Graph***, ***Triple Graph Morphism***, ***Triple Rule*** and ***Triple Transformation Step*** are brought in as they were introduced in [Sch94], [EEH+08] and [EEE+07]. Further ***Model Transformation*** follows: before the final definition of ***Model Transformation*** is given, its explained how to generate ***source-/forward- rules*** from ***triple rules***, what is ***match consistency*** and the theorem of ***canonical composition / decomposition*** is brought in. Similarly the ***Model Integration*** is described. As introduced in [EEE+07] first the ***source-/target-*** and ***integrations rule*** are put up, then, as it was for ***Triple Transformation,*** the ***canonical composition / decomposition*** is brought in, concluding with the definition of ***Model Integration.***

Although the theory of triple graph transformation / integration,describes triple graph transformation based on non-deleting triple rules, it appeared reasonable to build up a construct which would offer the possibility of extending the implementation by the use of deleting-rules. This intend led to the idea of a ***DPO triple rule*** described in Section II. 1.5. .

The examples for visualisation of explained approaches and techniques are realized in Class-Diagram language on left sides of the triple rules and Entity-Relationship-Diagram language on right sides respectively. It is assumed that this modelling languages are either wide known or can be intuitively understood from context. This modelling techniques are widely described in [SE+07] and [BD+04].

1.1. Triple Graph.

Understanding of the model transformation / integration based on triple graph grammars requires being familiar with the general graph theory and the triple graph / triple graph morphism concept. Therefore at this point the definition of graph, graph morphism, triple graph and triple graph morphism are brought in, followed by a realisation of the theoretical concepts in a practical example.

Definition 1 (Graph and Graph Morphism)

A graph $G=(V, E, s, t)$ *consists of a set* V *of nodes (also called vertices),* E *of edges and two functions* $src, tar: E \to V$ *,the* ***source*** *and* ***target*** *functions.*

Given graphs G_1, G_2 *with* $G_i=(V_i, E_i, src_i, tar_i)$ *for* $i=1,2$ *, a* ***graph morphisms*** $f: G_1 \to G_2$, $f=(f_V, f_E)$, *consists of two functions* $f_V: V_1 \to V_2$ *and* $f_E: E_1 \to E_2$ *that preserve the source and target functions, i.e.* $f_V \circ src_1 = src_2 \circ f_E$ *and* $f_V \circ tar_1 = tar_2 \circ f_E$.

Definition 2 (Triple Graph and Triple Graph Morphism)

Three graphs SG, CG *and* TG, *called* ***source***, ***connection***, *and* ***target*** *graphs, together with two graph morphisms* $s_G: CG \to SG$ *and* $t_G: CG \to TG$ *form a* ***triple graph*** $G=(SG \xleftarrow{s_G} CG \xrightarrow{t_G} TG)$. G *is called empty, if* SG, CG *and* TG *are empty graphs.*

A ***triple graph morphism*** $m=(s, c, t): G \to H$ *between two triple graphs* $G=(SG \xleftarrow{s_G} CG \xrightarrow{t_G} TG)$ *and* $H=(SH \xleftarrow{s_H} CH \xrightarrow{t_H} TH)$ *consists of three graph morphisms* $s: SG \to SH$, $c: CG \to CH$ *and* $t: TG \to TH$ *such that* $s \circ s_G = s_H \circ c$ *and* $t \circ t_G = t_H \circ c$. *It is injective, if morphisms s, c and t are injective.*

Image 1: Graphs: G_1 *and* G_2

On the Image 1 are two simple Graphs: G_1 in Class-Diagram language with only one node “a” of type “Class” and G_2 in Entity-Relationship language with only node “c” of type “Table”. According to Definition 1 (Graph and Graph Morphism) these graphs can be defined as fallows: $G_1=(V_1, E_1, s_1, t_1)$ and $G_2=(V_2, E_2, s_2, t_2)$, where $V_1=\{a: Class\}$, $E_1=\emptyset$, $s_1=\emptyset$, $t_1=\emptyset$ and $V_2=\{c: Table\}$, $E_2=\emptyset$, $s_2=\emptyset$, $t_2=\emptyset$.

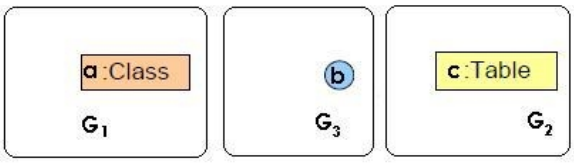


Image 2: Graphs: G_1, G_2 *and* G_3

In case, for instance, the diagrams on given images are used to model a data base representation in some object oriented programming language, it might be needed to describe the relationship (coexistence property) between nodes a and c. Since G_1 and G_2 are expected to stay in their own modelling language, connecting them directly with an edge will lead to major difficulties in later work. Besides, the intent of the theory is not only to establish a connection, but also a relation between the graphs. Which makes it a better solution in this case to insert a third graph G_3 (Image 2) into the diagram, which on the one hand represents the connection between G_1 and G_2 but on the other hand is not bound to any particular modelling language. A formal definition for graph G_3 is: $G_3=(V_3, E_3, s_3, t_3)$, with $V_3=\{b\}, E_3=\emptyset, s_3=\emptyset, t_3=\emptyset$.

The actual connection between a,b and c remains missing. To realize it, firstly, the three separate Graphs (G_1, G_2, G_3) are united into one triple graph G_4 which includes them all and for this reason can include also relations (visualised as edges) between nodes from own under-graphs; secondly the actual edges are created by embedding two morphisms $s_{13}: G_3 \to G_1$ and $t_{32}: G_3 \to G_2$. According to Definition 1 (Graph and Graph Morphism) this morphisms can be defined as: $s_{13}=(s_{V13}, s_{E13})$, where $s_{V13}: \{b\} \to \{a:Class\}$, $s_{E13}: \emptyset \to \emptyset$ and $s_{32}=(s_{V32}, s_{E32})$, where $s_{V32}: \{b\} \to \{c:Table\}$, $s_{E32}: \emptyset \to \emptyset$ respectively. The resulting construct is visualised in Image 3.

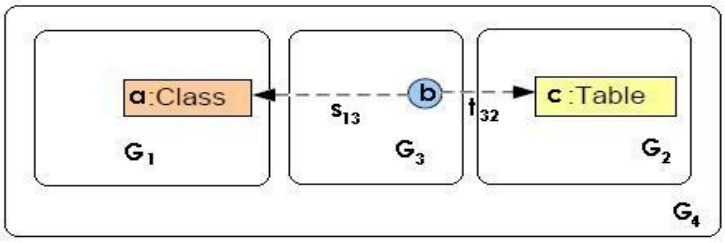


Image 3: Triple Graph: $G_1, G_2, G_3 \subseteq G_4$, morphisms s_{13}, t_{32}

In the end the graph G_4 can be defined as a triple graph according to Definition 2 (Triple Graph and Triple Graph Morphism): $G_4=(G_1 \xleftarrow{s_{13}} G_3 \xrightarrow{t_{32}} G_2)$, with G_1 - source graph, G_2 - target and G_3 - connection graphs, s_{13} - source and t_{32} - target morphisms.

1.2. Triple Rule

The ability to manually define a connection between two models is not the final aim of graph transformation / integration. The aim is a rule-based construction which can be automated. Continuing the example from section 1.1 a requirement case is shown and the solution of it via a triple rule and a triple transformation step – two concepts which make a automation possibility more near:

The requirement: to build up a data base representation in some programming language. The approach chosen: to proceed with a stepwise build-up of models representing the data base (entity relationship diagram) on source side and model representing some object oriented programming language (class diagram) on target side. Furthermore, every build-up step is defined by a unique rule. This rule describes changes to be made in every sub-graph of a triple graph to fulfil a transformation and is constructed in following way: the left-hand side of the rule - a triple graph which includes all components needed for the rule to be applied, the right-hand side - a triple graph, used to replace the left-hand side and a triple morphism, used to describe the replacement procedure of left-hand side triple graph with the right-hand one. Since chosen approach creates a build-up sequence – only non-deleting triple rules are used. As a result the triple morphism is injective and left-total.

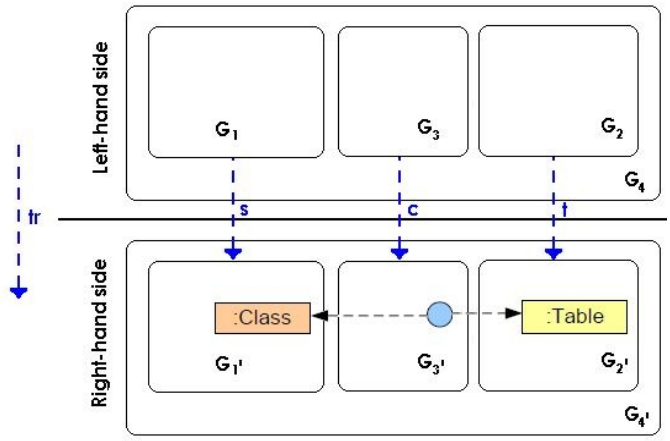


Image 4: Class2Table rule

Image 4 Illustrates a simple triple rule “Class2Table” which inserts a new node of type “Class” into the source part of a model, a new node of type “Table” into the target part and a connection node into the connection graph with two connection morphisms respectively. The left-hand side is an empty triple graph (in other words – there are no preconditions in this rule), the right-side is the result-graph including the new nodes (of types “Class”,”Connection”,”Table”). To describe the transformation procedure, a triple morphism $tr=(s,c,t):G_4\rightarrow G_{4'}$ is needed. It consists of three separate morphism. Each of this morphisms is used to picture one of the transformations: $s:G_1\rightarrow G_{1'}$, $c:G_3\rightarrow G_{3'}$ and $t:G_2\rightarrow G_{2'}$.

Definition 3 (Triple Rule tr and Triple Transformation Step)

*A **triple rule tr** consists of triple graphs L and R, called **left-hand** and **right-hand sides**, and an injective triple graph morphism* $tr=(s,c,t):L\to R$.

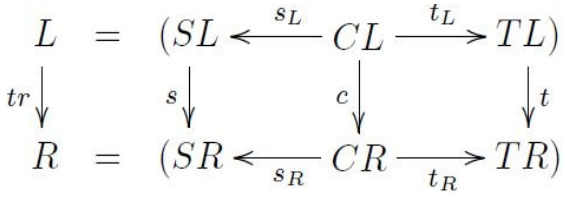


Image 5: Triple Rule tr

Given a triple rule $tr=(s,c,t):L\to R$, *a triple graph G and a triple graph morphism* $m=(sm,cm,tm):L\to G$, *called **triple match m**, a **triple graph transformation step** (TGT-step)* $G \overset{tr,m}{\Rightarrow} H$ *from G to a triple graph H is given by three pushouts ([EEP+06] "Graphs, Typed Graphs, and the Gluing Construction.")* (SH,s',sn), (CH,c',cn) *and* (TH,t',tn) *in category Graph with induced morphisms* $s_H:CH\to SH$ *and* $t_H:CH\to TH$.

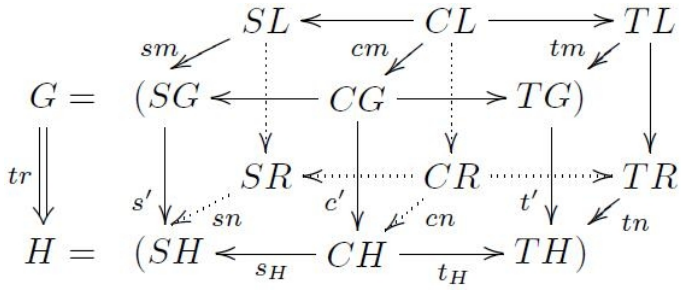


Image 6: Triple Transformation Step

Assuming to have already a model representing some existing data base (for instance a model already having two nodes of type "Class" with a node of type "Attribute" connected to one of them in the class-diagram part of the model, three connection nodes in the connection part and three nodes: two of type "Table", one of type "Column" respectively in the entity-diagram part – like in Image 9) and a triple rule with a node of type "Class" connected to a node of type "Table" in its left-hand side (Image 8), a further difficulty becomes obvious: in the example data base model there are two "class2table" structures, which correspond to the left-hand side of given rule.

The solution of this problem are three morphisms. Each of this morphisms defines an appropriate correspondence between one of sub-graphs in the left-hand side of the rule and the sub-graph of the example data base model. These morphisms are called matches.

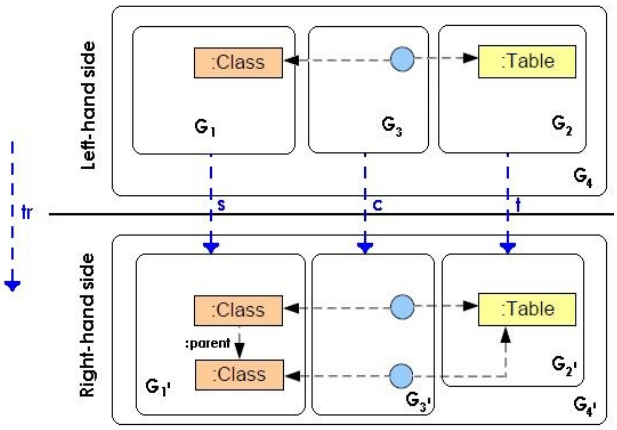


Image 8: Subclass2Table rule

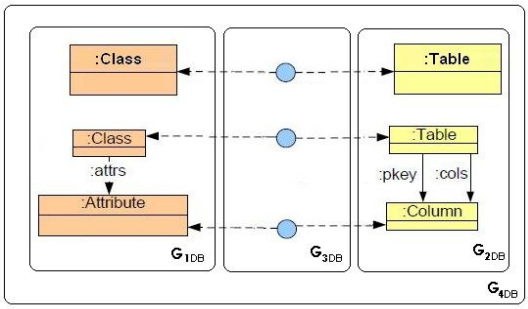


Image 9: Example DB

To have the matches between given rule and the example graph provides the possibility to perform the transformation step from Definition 3 (Triple Rule tr and Triple Transformation Step) by applying the rule to the example graph. The result is a triple graph $G_{4DB'}$ representing the example data base model, which is extended by the right-hand side of given rule in points of matches (Image 10).

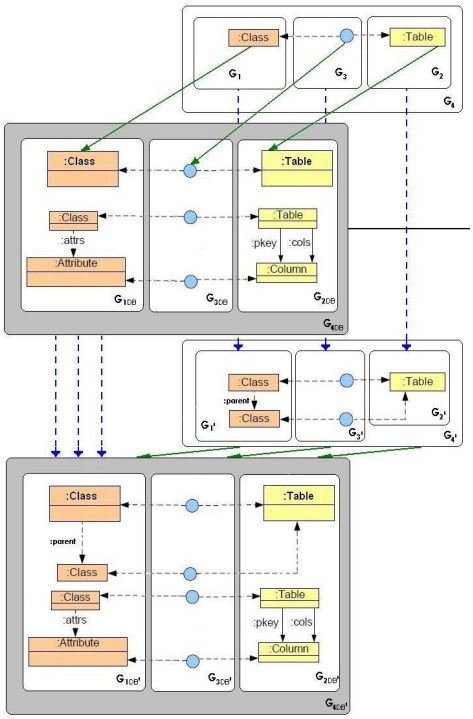


Image 10: Example: Performing transformation step

1.3. Model Transformation

Model transformation is another realisation of "divide and conquer" principle. To perform a transformation from one model into an other, rules for synchronous creation of these models are defined. These are simple triple rules (according to Definition 3). All nodes from given models should be reachable by the rules. It means there shouldn't be a node in (for example) the target model which wouldn't appear in right-hand side of some rule.

As soon as transformation (connection) rules are defined, the actual triple graph transformation process can start. First, the source model is parsed in order to find a sequence of rules, which were used to build it up. In the beginning only the source model is available and original triple rules, which are defined for the transformation, can't be used during establishment of the source sequence (parsing of the source model can be performed only with triple rules containing source graphs, but original triple rules contain also connection and target graphs). For that reason original triple rules are reduced to source triple rules, which are same triple rules with connection and target graphs empty. Yet, for later phases connection and target parts will be needed, so for every source rule there is a unique forward rule. Forward rules are constructed in such a way that after applying them to a source model (a triple graph in which only the source graph is not empty), only connection and target graphs of the model change but the source graph stays same.

Definition 4 (Derived Triple Rules)

Given a triple rule tr as in Definition 3 (Triple Rule tr and Triple Transformation Step), a source rule tr_S and a forward rule tr_F can be constructed as shown in Image 11:

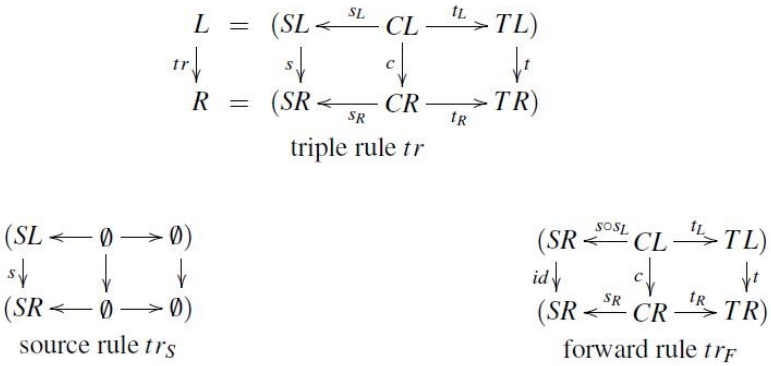


Image 11: Construction of source- and forward- rules

After establishing the source sequence, the corresponding source model is build up by applying source rules in inverse order of the parsing sequence and by using comatches as new matches. Then forward rules are applied in the same order. Yet again a difficulty occurs: As already mentioned in the description of triple rules application example (1.2. Triple Rule) there are often multiple possibilities to apply same triple rule to same triple graph. Which of these possibilities is selected depends on chosen matches. Therefore in order to perform a consistent graph transformation, forward rules are applied strictly with the notion of match consistency.

Definition 5 (Match Consistency)

Let tr_S^* *and* tr_F^* *be sequences of source rules* tri_S *and forward rules* tri_F*, which are derived from same rule tri for* $i=1,\ldots,n$*. Let further* $G_{00} \overset{tr_S^*}{\Rightarrow} G_{n0} \overset{tr_F^*}{\Rightarrow} G_{nn}$ *be a TGT-sequence with* (mi_S, ni_S) *being match and comatch of* tri_S *(respectively* (mi_F, ni_F) *for* tri_F*) then match consistency of* $G_{00} \overset{tr_S^*}{\Rightarrow} G_{n0} \overset{tr_F^*}{\Rightarrow} G_{nn}$ *means that S-component of the match* mi_F *is uniquely determined by the comatch* ni_S $(i=1,\ldots,n)$*.*

Theorem 1 (Canonical Decomposition and Composition Result - Forward Rule Case)

1. ***Decomposition****: For each TGT-sequence based on triple rules tr**
   (1) $G_0 \overset{tr^*}{\Rightarrow} G_n$ *there is a canonical match consistent TGT-sequence*
   (2) $G_{00}=G_{00} \overset{tr_S^*}{\Rightarrow} G_{n0} \overset{tr_F^*}{\Rightarrow} G_{nn}=G_n$ *based on corresponding source rules* tr_S^* *and forward rules* tr_F^**.*
2. ***Composition****: For each match consistent transformation sequence (2) there is a canonical transformation sequence (1).*
3. ***Bijective Correspondence****: Composition and Decomposition are inverse to each other.*

Based on Theorem 1 (for prove see [EEE+07]), finally the formal definition of model transformation can be introduced. According to that definition, model transformation is a triple of a source graph, a target graph and a forward rule sequence. In this triple the source component of first triple graph in given sequence is the given source graph, the target component of last triple graph in given sequence is given target graph. Such a transformation is called source consistent, if for the forward sequence a source sequence exist and their combination is match consistent.

Definition 6 (Model Transformation)

A ***model transformation sequence*** $(G_S, G_1 \overset{tr_F^*}{\Rightarrow} G_n, G_T)$ *consists of a source graph* G_S*, a target graph* G_T*, and a source consistent forward TGT-sequence* $G_1 \overset{tr_F^*}{\Rightarrow} G_n$ *with* $G_S=proj_S(G_1)$ *and* $G_T=proj_T(G_n)$*.*

Source consistency *of* $G_1 \overset{tr_F^*}{\Rightarrow} G_n$ *means that there is a source transformation sequence* $\emptyset \overset{tr_S^*}{\Rightarrow} G_1$*, such that* $\emptyset \overset{tr_S^*}{\Rightarrow} G_1 \overset{tr_F^*}{\Rightarrow} G_n$ *is match consistent. A model transformation* $MT: VL_{S0} \equiv> VL_{T0}$ *is defined by model transformation sequences* $(G_S, G_1 \overset{tr_F^*}{\Rightarrow} G_n, G_T)$ *with* $G_S \in VL_{S0}$ *and* $G_T \in VL_{T0}$*.*

Remark 1: *A model transformation* $MT: VL_{S0} \equiv> VL_{T0}$ *is a relational dependency and only in special cases a function. This allows to show that* $MT: VL_{S0} \equiv> VL_{T0}$ *defined above is in fact* $MT: VL_S \equiv> VL_T$*.*

Fact 1 (Syntactical Correctness of Model Transformation MT): *Given* $G_S \in VL_{S0}$ *and* $G_1 \overset{tr_F^*}{\Rightarrow} G_n$ *source consistent with* $proj_S(G_1) = G_S$, *then* $G_T = proj_T(G_n) \in VL_T$ *and* $G_S \in VL_S$, *i.e.* $MT : VL_S \equiv > VL_T$. *(for prof refer to [EEH+08])*

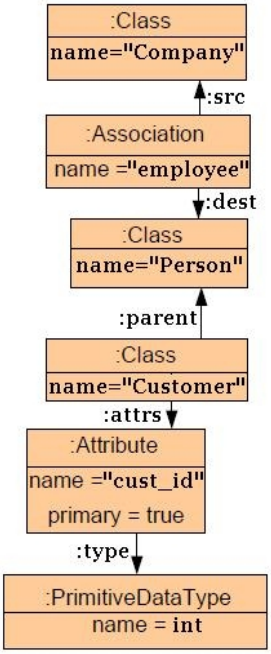


Image 13: Transformation example source graph

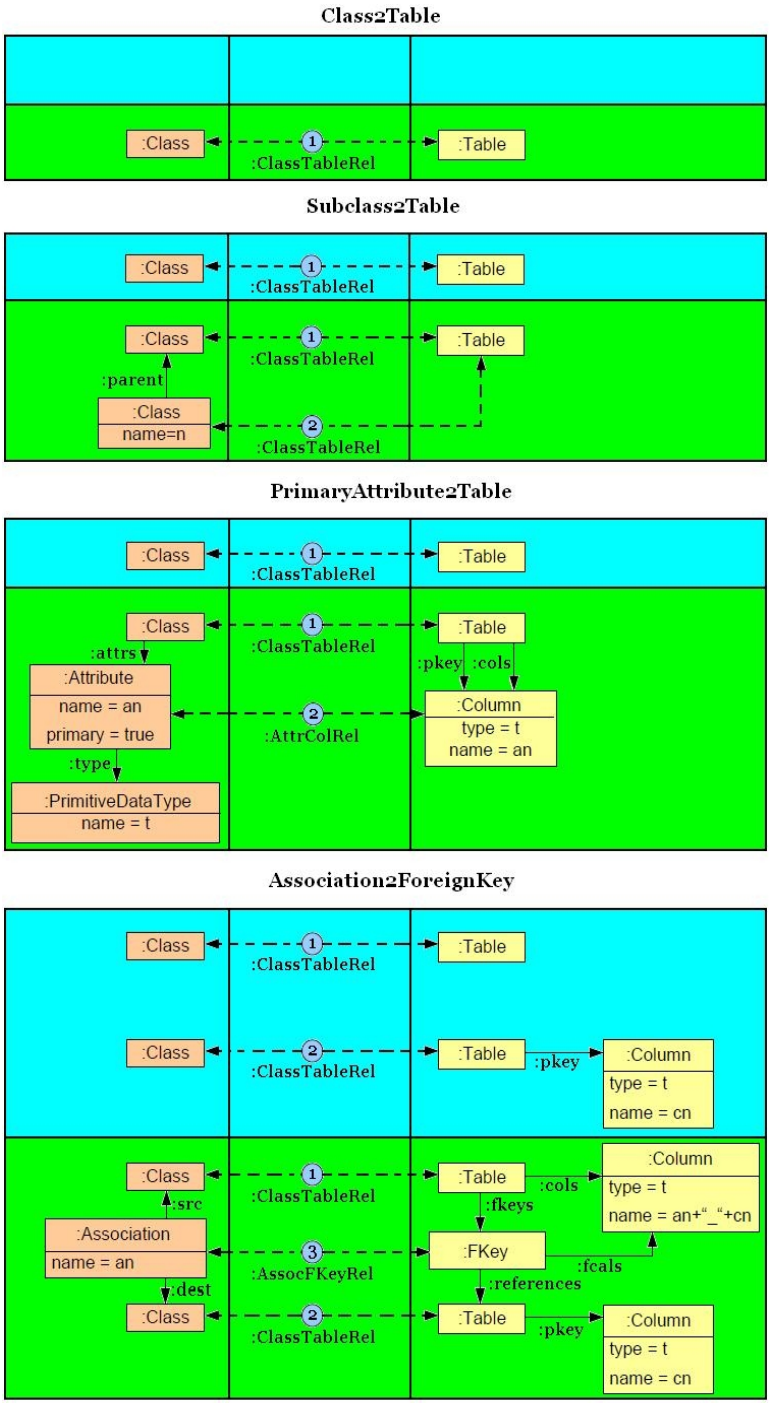


Image 12: Transformation example rules

build-up: first row (light blue) – left-hand side triple graph, second row (green) – right-hand side triple graph

According to Definition 6, to illustrate the process of triple graph transformation procedure a source model is needed, a set of triple rules and a sequence of corresponding forward rules, in the end of which the target graph corresponds the actual target model.

Let the model in Image 13 be the source model and the set of rules in Image 12 - the given set of transformation rules (Image 12). Now it is possible to generate corresponding source and forward rules. Source rules are created by deleting connection and target components from original rules (Image 15).

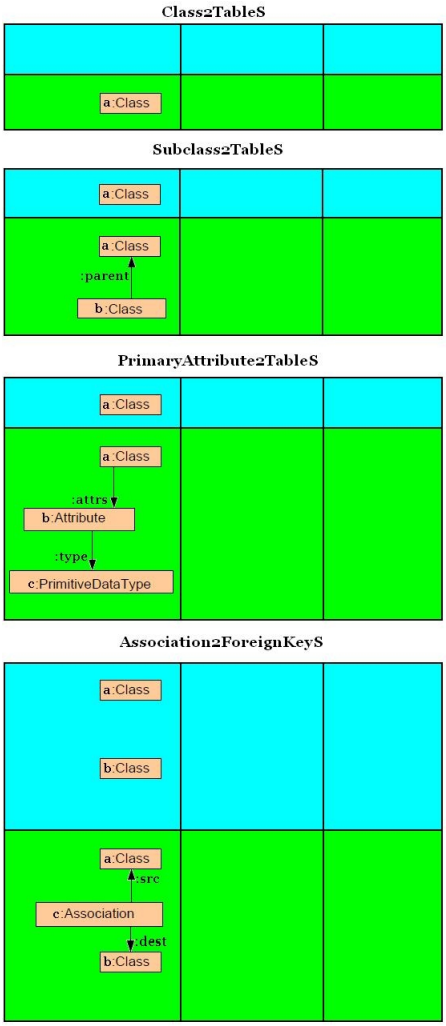


Image 15: Example source rules

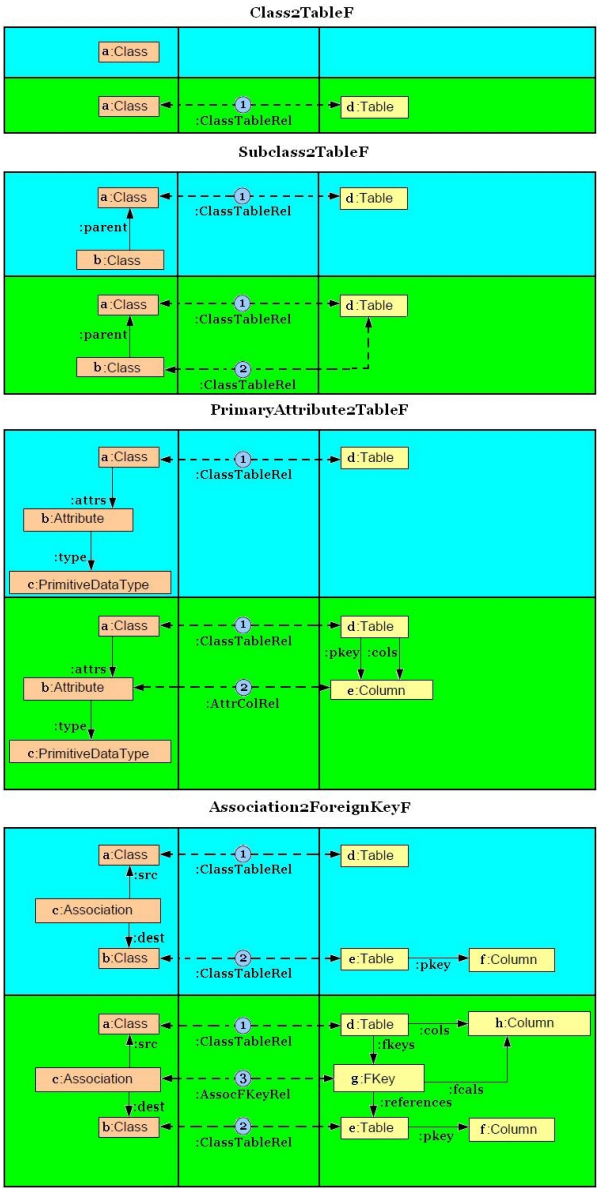


Image 14: Example forward rules

A forward rule is applied to a triple graph, which normally has a not-empty source graph with the intend of constructing corresponding connection- and target- graphs, without changing the source graph. Therefore in the source part of forward rules between left- and right- hand triple graphs, an identity morphism is needed. Yet an identity morphism alone is not enough, since in connection- and target- parts of the rule simple injective morphisms are used. If changes to original rules would be limited to replacement of the morphism, connecting left- and right- hand source graphs, with an identity morphism, the result triple graph in right-hand side contained often a partial morphism from connection- to source- graphs, which contradicts to Definition 2 (Triple Graph and Triple Graph Morphism). Therefore the construction of a forward rule according to Definition 4 (Derived Triple Rules) contains also the replacement of the morphism from connection- to source- graphs of left-hand side triple graph in the rule with the concatenation of original morphism from connection- to source- graphs of left-hand side triple graph (s_L in Image 11) with original morphism transforming left-hand side source graph into right-hand one (s in Image 11). As a result of this action, the left-hand source graph contains nodes and edges created by s_L and s (since s is non-deleting by Definition 3). It means the left- and right- hand source graphs of forward rules are equal to each other and to the right-hand source graph from original rule (Image 12 triple rules, Image 14 corresponding forward rules).

Having source rules, the source sequence which led to the source model can be established. In most cases it is done manually. The theory implicates the source model have been build up by the use of some rules, from which given transformation rules were derived. So the source sequence is supposed to exists. Yet often it is not the case. Then the source sequence is established brute-force manually, or automated. In III. 3.10. a concept realisation is introduced to shrink the search tree of brute-force pattern search, by the use of DPO rules (1.5.) and backtracking.

In this example it is assumed that the source sequence $G_0 \overset{Class2TableS}{\Rightarrow} G_1 \overset{Class2TalbeS}{\Rightarrow} G_2 \overset{Subcalss2TableS}{\Rightarrow} G_3 \overset{PrimaryAttribute2ColumnS}{\Rightarrow} G_4 \overset{Association2ForeignKeyS}{\Rightarrow} G_5$ was established manually according to Table 1.

After the source sequence is established, forward rules can be applied to the source model in source sequence corresponding order with the notion of match consistency according to Table 2. The target graph of triple graph G_{10} is the actual target model (Image 16) and the graph transformation can be formally noted as

$$(proj_S(G_5), G_5 \overset{Class2TableF}{\Rightarrow} G_6 \overset{Class2TalbeF}{\Rightarrow} G_7 \overset{Subcalss2TableF}{\Rightarrow} G_8 \overset{PrimaryAttribute2ColumnF}{\Rightarrow} G_9 \overset{Association2ForeignKeyF}{\Rightarrow} G_{10}, proj_T(G_{10})).$$

Moreover, the example graph transformation is source consistent:

1. The existence of a source sequence is a fact by source model construction (Table 1).
2. Match consistency of the source sequence can be shown by comparing comatcha source components from Table 1 with matches source components from Table 2 - Table 3.

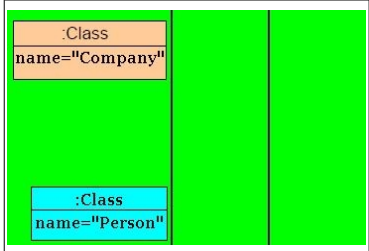
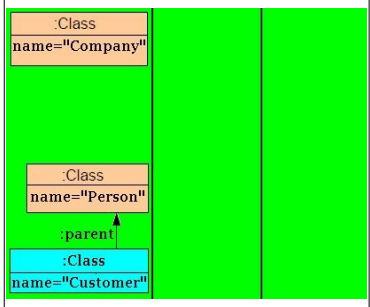
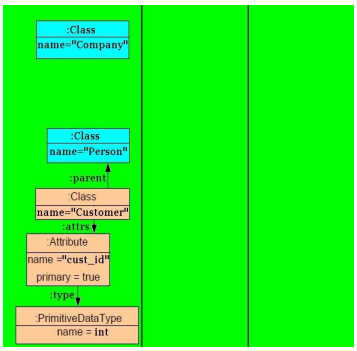
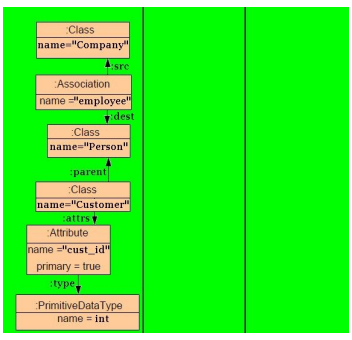
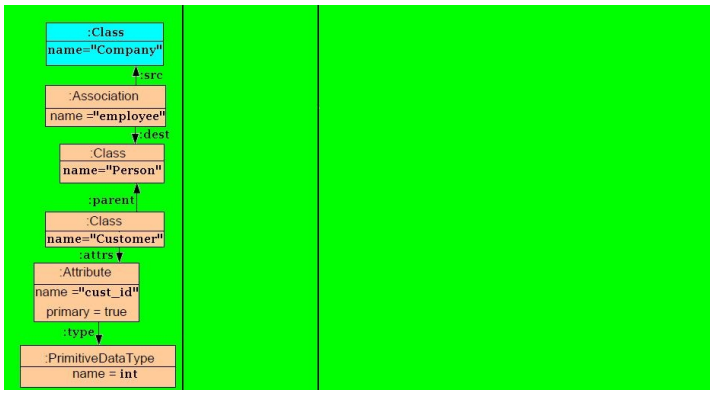
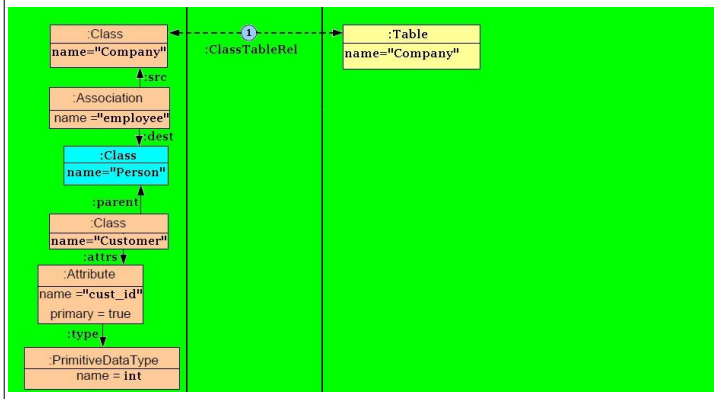
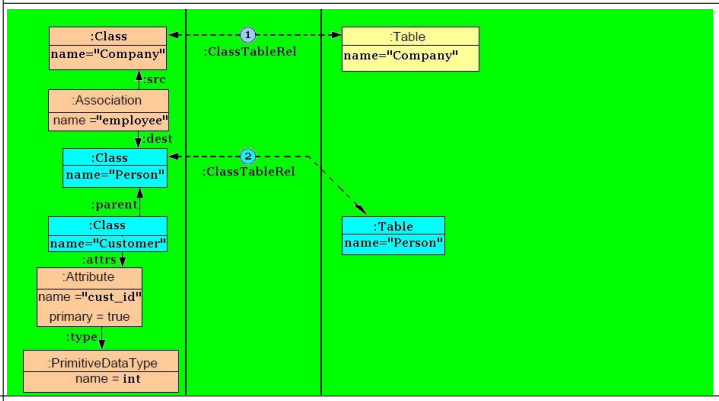
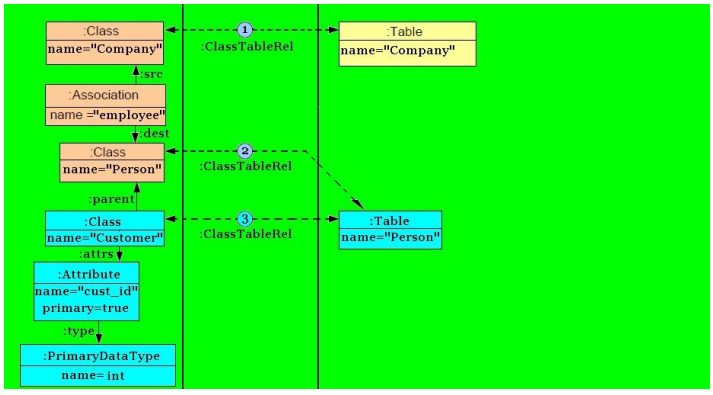
Rule	Visual representation	Name	match **n**	Comatch **n-1** **(only source part)**	Comment
Class2TableS		G_0	$\emptyset$		Class2Table is applied with an empty match.
Class2TableS		G_1	$\emptyset$	**“a” - “Company”**	Again applying Class2Table with empty match.
Subclass2TableS		G_2	**“a” - “Person”**	**“a” - “Person”**	Left hand triple graph source graph from rule Subclass2Table is a graph with one node of type “Class”, therefore choosing one node of type “Class” - from G_2 - “Person”(the one being parent to an other node of type “Class” in Image 13).
PrimaryAttribute2ColumnS		G_3	**“a”-”Customer”**	**“a” - ”Person”** **“b” - ”Customer”**	Precondition (left hand triple graph source graph) of rule PrimaryAttribute2Column is also a graph with one node of type “Class” . Yet according to Image 13 - “Customer” is the node, which has a primary attribute.
Association2ForegnKeyS		G_4	**“a”-”Company”** **“b”-”Person”**	**“a” - ”Customer”** **“b” - “Attribute”** **“c” -** **”PrimitiveDataType”**	In the precondition of Association2ForegnKey there are two nodes of type “Class”: Therefore two nodes type “Class” to choose for G_4 . Since in Image 13 “Person” and “Company” are connected with a node of type “Association”, those are chosen.
		G_5		**“a”-”Company”** **“b”-”Person”** **“c”-”Association”**	G_5 corresponds to the example source graph from Image 13.

Table 1: Example source sequence execution

(Light blue nodes in “Visual representation” are ones matched with nodes in left hand side of rule *beginning* at same raw with corresponding “Visual representation”; the “co-match” entry belongs to rules *ending* in same raw with corresponding “co-match”)

Rule		Visual representation	Name	match **n**
Class2TableF			G_5	**Source:** **"a" - "Company"** **Connection:** **Ø** **Target:** **Ø**
	Class2TableF		G_6	**Source:** **"a" - "Person"** **Connection:** **Ø** **Target:** **Ø**
Subclass2TableF	Class2TableF		G_7	**Source:** **"a" - "Person"** **"b" - "Customer"** **Connection:** **"1" - "2"** **Target:** **"d" - "Person"**
	PrimaryAttribute2ColumnF		G_8	**Source:** **"a" - "Customer"** **"b" - "Attribute"** **"c" - "PrimaryDataType"** **Connection:** **"1" - "3"** **Target:** **"d" - "Person"**

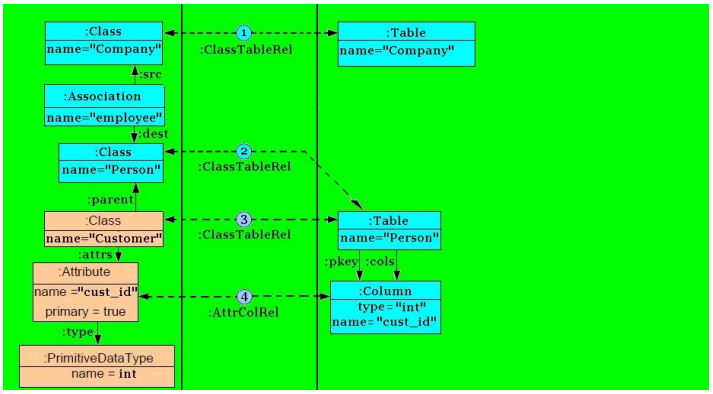
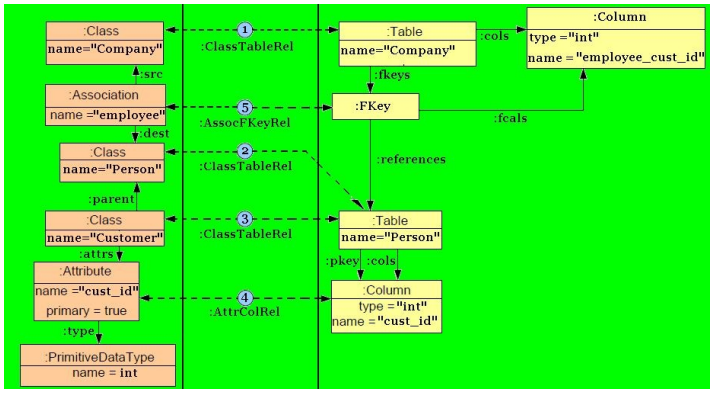
Association2ForegnKeyF		G_9	**Source:** **“a”-”Company”** **“b”-”Person”** **“c”-”Association”** **Connection:** **“1” - ”1”** **“2” - “2”** **Target:** **“d” - “Company”** **“e” - “Person”** **“f” - “cust_id”**
		G_{10}	

Table 2: Example forward sequence execution

(Light blue nodes in “Visual representation” are ones matched with nodes in left hand side of rule *beginning* at same raw with corresponding “Visual representation”)

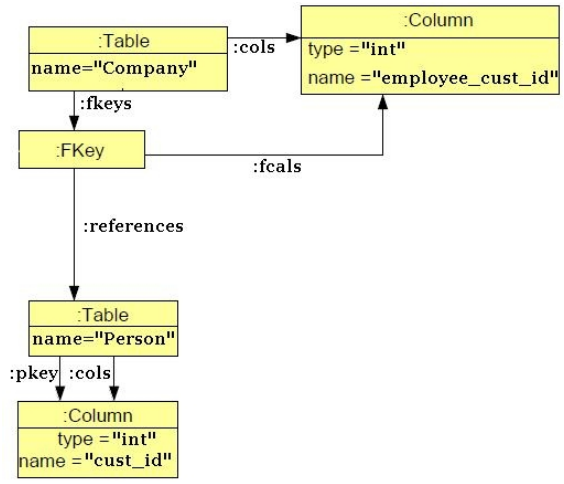


Image 16: Transformation example target graph

Source comatch **n (1..5)**	Forward match **n (6..10)**
“a” - “Company”	**“a” - “Company”**
“a” - “Person”	**“a” - “Person”**
“a” - ”Person” **“b” - ”Customer”**	**“a” - “Person”** **“b” - “Customer”**
“a” - ”Customer” **“b” - “Attribute”** **“c” - ”PrimitiveDataType”**	**“a” - ”Customer”** **“b” - “Attribute”** **“c” - ”PrimitiveDataType”**
“a”-”Company” **“b”-”Person”** **“c”-”Association”**	**“a”-”Company”** **“b”-”Person”** **“c”-”Association”**

Table 3: Example match consistency check

1.4. Model Integration

The concept and realisation of model integration resembles the concept and realisation of model transformation, yet it aims another task specification. Model transformation is meant for building up a new model in some different modelling language, which would correspond to an existing one. An example use case is modelling a new data base for an existing object oriented application.

But what can be done if both data base and application already exist? Model transformation fails in such a case, since there is no need in new creation of the second model. The task specification in this case is to build up a connection between two existing models. This is the task of model integration.

Similarly to the model transformation, the process of model integration begins with the triple rule definition. After that the two models are parsed for a triple rule sequence, which was used to build them up. Yet since in this case not only source, but also target models are known, the sequence searched is not a source, but a source-target sequence. This means that the triple rules used to generate this sequence are source-target rules. So, first all rules are transformed into source-target rules by deleting the connection graphs from the original rules. However, like in the model transformation case, further the connection graphs will be needed. To restore connection graphs in existing source-target models integration rules are created. When an integration rule is applied to a triple graph, source and target graphs stay same and only connection graphs are changed.

Definition 7 (Source-target rule, integration rule)

Given a triple rule tr as in Definition 3 (Triple Rule tr and Triple Transformation Step), a ***source- target rule*** tr_{ST} *and an* ***integration rule*** tr_I *can be constructed as shown below:*

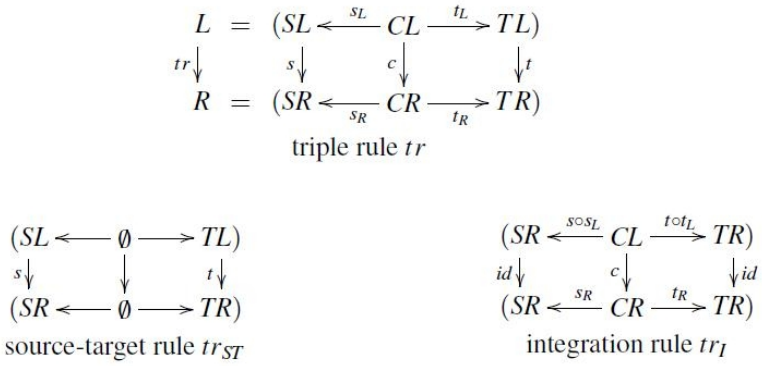


Image 17: Construction source-target and integration rules

Model Integration includes also a theorem for canonical composition and decomposition (for formal proof refer to [EEH+08]), which states that a triple rules sequence can be converted into a source-target rule sequence followed by an integration rule sequence and visa versa. This observation makes it possible after finding a source-target rule sequence which leads to the building up of given source and target models, to apply integration rules in corresponding order and build up the connection graph in that way.

Theorem 2 (Canonical Decomposition and Composition Result - Integration Rule Case)

1. ***Decomposition:*** *For each TGT-sequence based on triple rules* tr^*

   (1) $G_0 \overset{\mathrm{tr}^*}{\Rightarrow} G_n$ *there is a canonical S-T -match consistent TGT-sequence*

   (2) $G_{00} = G_{00} \overset{\mathrm{tr}^*_{ST}}{\Rightarrow} G_{n0} \overset{\mathrm{tr}^*_{I}}{\Rightarrow} G_{nn} = G_n$ *based on corresponding source-target rules* tr^*_{ST} *and integration rules* tr^*_{I}.

2. ***Composition:*** *For each S-T -match consistent transformation sequence (2) there is a canonical transformation sequence (1).*

3. ***Bijective Correspondence:*** *Composition and Decomposition are inverse to each other.*

Finally having the source and target models, the integration sequence and the final triple graph with connection graph filled, the formal definition of Model integration can be verbalised (Definition 8). Having this two models and a sequence given, the corresponding connection graph can be build up. The integration sequence itself is match consistent.

Additionally, as Remark 2 states – there exists only one unique source-target sequence for a given integration sequence, since the matches and comatches, which are used to apply source-target rules, are determined by the match in the integration rules sequence.

Definition 8 (Model Integration)

A model integration sequence $((G_S, G_T), G_0 \overset{\mathrm{tr}^*_{I}}{\Rightarrow} G_n, G)$ *consists of a source and a target model* G_S *and* G_T*, an integrated model G and a source-target consistent TGT-sequence* $G_0 \overset{\mathrm{tr}^*_{I}}{\Rightarrow} G_n$ *with* $G_S = proj_S(G_0)$ *and* $G_T = proj_T(G_0)$*. Source-target consistency of* $G_0 \overset{\mathrm{tr}^*_{I}}{\Rightarrow} G_n$ *means that there is a source-target transformation sequence* $\emptyset \overset{\mathrm{tr}^*_{ST}}{\Rightarrow} G_0$*, such that* $\emptyset \overset{\mathrm{tr}^*_{ST}}{\Rightarrow} G_0 \overset{\mathrm{tr}^*_{I}}{\Rightarrow} G_n$ *is match consistent. A model integration* $MI : VL_{S0} \times VL_{T0} \equiv > VL$ *is defined by model integration sequences* $((G_S, GT), G_0 \overset{\mathrm{tr}^*_{I}}{\Rightarrow} G_n, G)$ *with* $G_S \in VL_{S0}$, $G_T \in VL_{T0}$ *and* $G \in VL$.

Remark 2: *Given model integration sequence* $((G_S, G_T), G_0 \overset{\mathrm{tr}^*_{I}}{\Rightarrow} G_n, G)$ *the corresponding source-target TGT-sequence* $\emptyset \overset{\mathrm{tr}^*_{ST}}{\Rightarrow} G_0$ *is uniquely determined. The reason is that each comatch of* tri_{ST} *is completely determined by S- and T-component of the match of* tri_I*, because of embedding* $\mathrm{R}(tri_{ST}) \equiv > \mathrm{L}(tri_I)$*. Furthermore, each match of* tri_{ST} *is given by uniqueness of pushout complements along injective morphisms with respect to non-deleting rule* tri_{ST} *and its compatch. Moreover, the source-target TGT-sequence implies* $G_S \in VL_{S0}$ *and* $G_T \in VL_{T0}$.

To visualise a model integration, source and target models and a set of triple rules are needed. Let the source graph and triple rules be the same as in the transformation example (Image 12 and Image 13). Further, let the target graph from G_{10} (Image 16) be the target model.

Next source-target and integration rules have to be constructed out of triple rules. To construct source-target rules again a deletion has to be performed, yet this time only in connection graphs of given triple rules(Image 18).

To construct integration rules a similar procedure as in forward rules construction is performed, but this time in source and target parts of given triple rules. To ensure that source and target graphs stay untouched after source-target rule application to a triple graph, again identity morphisms are needed. To preserve consistency of right-hand graph in the rule, after an injective morphism execution in connection parts – a concatenation of morphisms is performed ($s \circ s_L$ and $t \circ t_L$ in Image 17). This concatenation leads to the source and target graphs in left-hand sides of the rules being equal to source and target graphs in right-hand sides of rules respectively (Image 19).

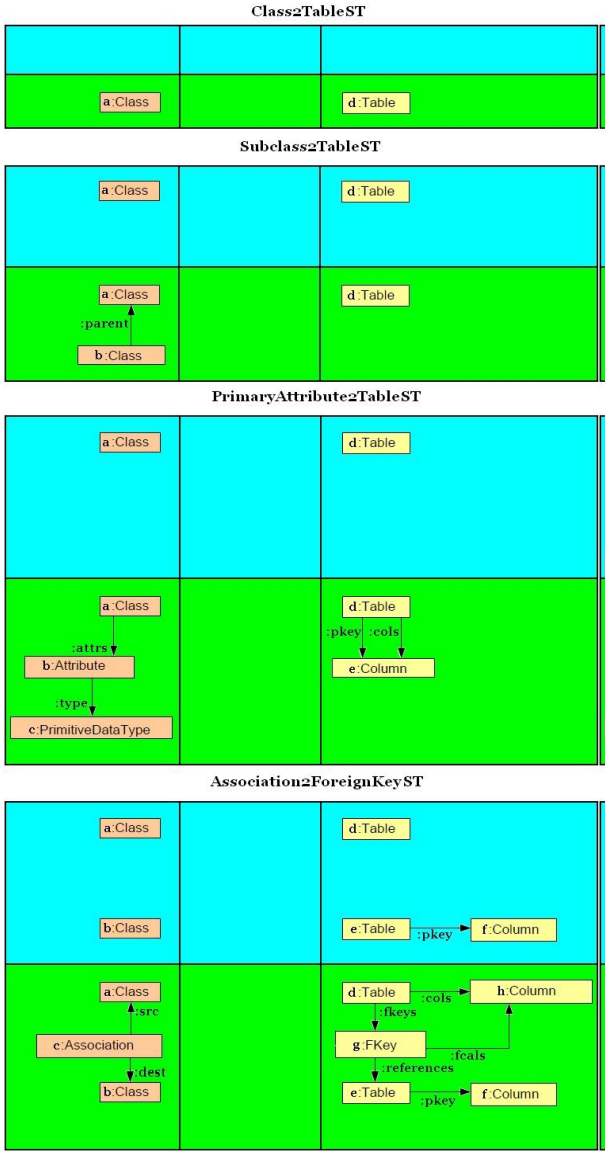
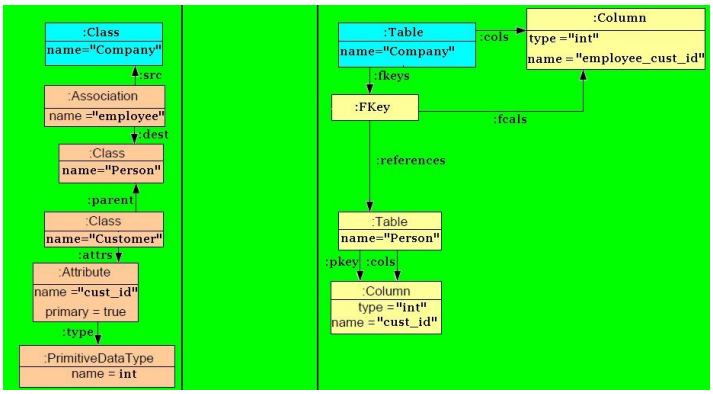
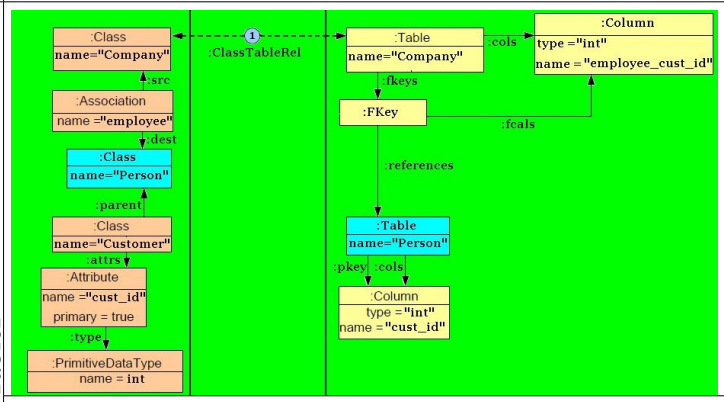
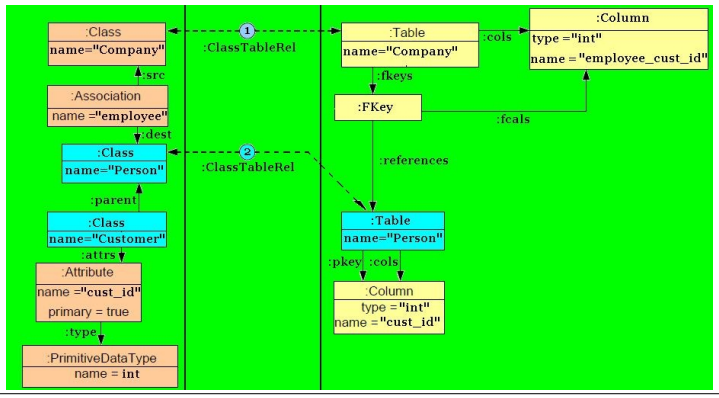


Image 18: Example source-target rules

Image 19: Example integration rules

Further the pattern search for the source-target sequence is performed brute-force manually or automated. Since the sequence of triple rules is known from transformation example: $G_0 \overset{Class2Table}{\Rightarrow} G_1 \overset{Class2Talbe}{\Rightarrow} G_2 \overset{Subcalss2Table}{\Rightarrow} G_3 \overset{PrimaryAttribute2Column}{\Rightarrow} G_4 \overset{Association2ForeignKey}{\Rightarrow} G_5$, and according to Theorem 2 - a canonical S-T consistent sequence of triple rules (which the sequence above is) can be decomposed to a corresponding source-target and integration rules sequence:
$G_0 \overset{Class2TableST}{\Rightarrow} G_1 \overset{Class2TalbeST}{\Rightarrow} G_2 \overset{Subcalss2TableST}{\Rightarrow} G_3 \overset{PrimaryAttribute2ColumnST}{\Rightarrow} G_4 \overset{Association2ForeignKeyST}{\Rightarrow} G_5$ and
$G_5 \overset{Class2TableI}{\Rightarrow} G_6 \overset{Class2TalbeI}{\Rightarrow} G_7 \overset{Subcalss2TableI}{\Rightarrow} G_8 \overset{PrimaryAttribute2ColumnI}{\Rightarrow} G_9 \overset{Association2ForeignKeyI}{\Rightarrow} G_{10}$ respectively. After executing the source-target sequence and gaining source-target graph G_5, the integration sequence can be applied as follows in Table 4.

Rule		Visual representation	Name	match **n**
Class2TableF			G_5	**Source:** **"a" - "Company"** **Connection:** $\emptyset$ **Target:** **"d" - "Company"**
	Class2TableF		G_6	**Source:** **"a" - "Person"** **Connection:** $\emptyset$ **Target:** **"d" - "Person"**
Subclass2TableF			G_7	**Source:** **"a" - "Person"** **"b" - "Customer"** **Connection:** **"1" - "2"** **Target:** **"d" - "Person"**

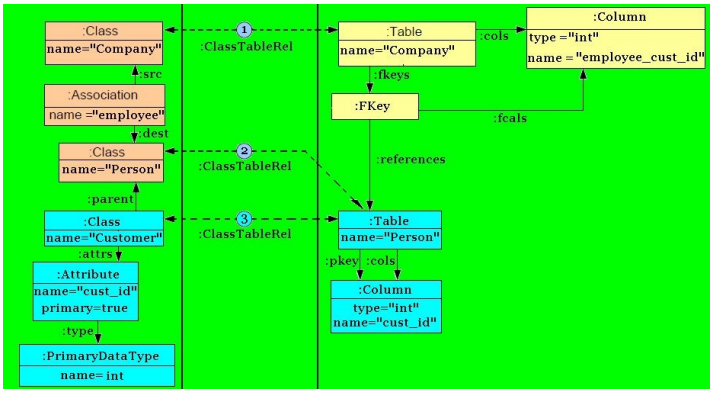
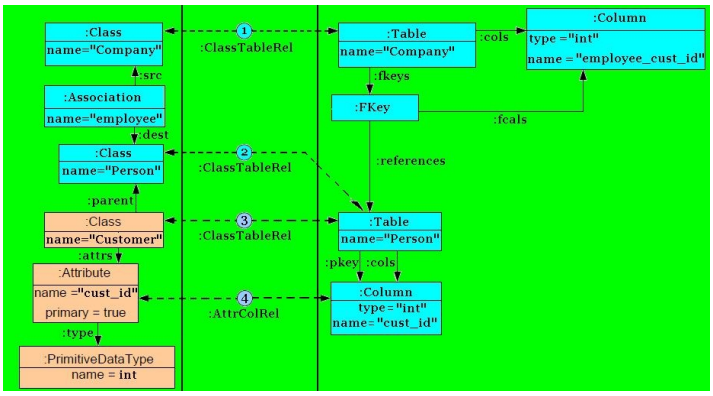
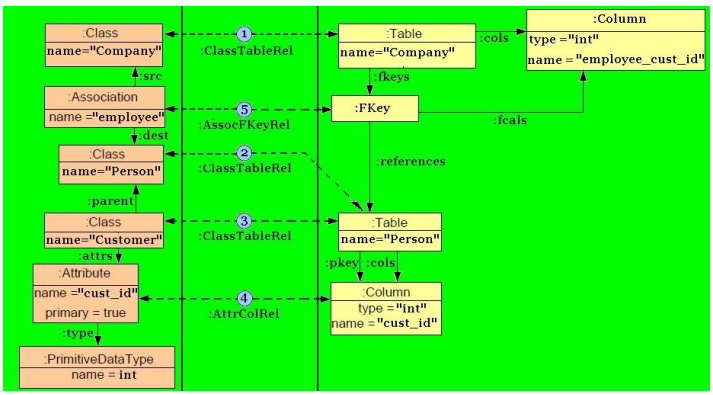
Association2ForegnKeyF	PrimaryAttribute2ColumnF		G_8	**Source:** **"a" - "Customer"** **"b" - "Attribute"** **"c" - "PrimaryDataType"** **Connection:** **"1" - "3"** **Target:** **"d" - "Person"** **"e" - "Person"**
			G_9	**Source:** **"a"-"Company"** **"b"-"Person"** **"c"-"Association"** **Connection:** **"1" - "1"** **"2" - "2"** **Target:** **"d" - "Company"** **"e" - "Person"** **"f" - "cust_id"** **"g" - ":FKey"** **"h" - "employee_cust_id"**
			G_{10}	

Table 4: Example integration sequence execution

(Light blue nodes in "Visual representation" are ones matched with nodes in left hand side of rule *beginning* at same raw with corresponding "Visual representation")

1.5. DPO Triple Rule

In process of implementation of the triple graph theory a need for the possibility to create deleting triple rules occurred. It is not possible to realize this approach by the use of an injective triple graph morphism between left- and right-hand triple graphs in the rule. This fact led to the extension of the theoretical concept of the triple rule.

The non-deletion property is important for triple graph transformation theory because it prevents triple graph transformation/-integration sequences ending up in loops, by adding and deleting same components of a graph. This is a temporal need. So, the task assignment was stated as follows: needed was a graph transformation mechanism, which by its limitation to injectivity, would form same result set as an injective triple morphism formed, yet by lifting that limitation creating a result set (triple rule right-hand side) extended by triple graphs, which would be subgraphs of left-hand side ones.

A solution for such a task is again a triple graph. Instead of using a triple-morphism in a triple rule specification three triple graphs are taken.

Definition 9 (DPO Triple Rule and Extended Transformation Step)

*A **DPO rule dtr** consists of triple graphs* $L=(L_S \xleftarrow{s_L} L_C \xrightarrow{t_L} L_T)$ *and* $R=(R_S \xleftarrow{s_R} R_C \xrightarrow{t_R} R_T)$ *called **left-hand** and **right-hand sides**, a triple graph* $K=(K_S \xleftarrow{s_K} K_C \xrightarrow{t_K} K_T)$ *and two inclusion triple graph morphisms* $tr_L=(kl_s, kl_c, kl_t): K \to L$ *and* $tr_R=(kr_s, kr_c, kr_t): K \to R$, *so that* $dtr=(L \xleftarrow{tr_L} K \xrightarrow{tr_R} R)$.

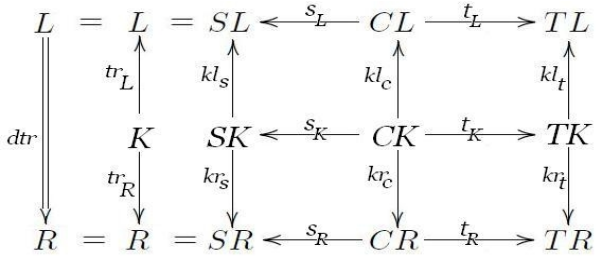


Image 20: DPO Rule dtr

Given a DPO rule $dtr=(L \xleftarrow{tr_L} K \xrightarrow{tr_R} R)$, *a triple graph G and a triple match* $m=(sm, cm, tm): L \to G$ *- which fulfil the gluing condition (according to [EEP+06] "Definition 6.3 (gluing condition in adhesive HLR systems)"), then the **extended triple graph transformation step (ETGT-step)*** $G \overset{dtr, m}{\Rightarrow} H$ *from G to a triple graph H (Image 22) is given as follows:*

- *POC SK', CK' and TK' are unique, because* kl_s, kl_c *and* kl_t *are monomorphisms and TripleGraph is a weak adhesive HLR category (see [EEP+06] "Fact 4.27 Theorem 4.26 Properties of (weak) adhesive HLR categories").*
- *POC SK', CK' and TK' exist, because sm, cm, tm fulfil the gluing condition by construction.*
- *PO objects SH, CH and TH exist uniquely by [EEP+06]"Fact 4.18 (TripleGraphs is an adhesive HLR category)" and [EEP+06]"Definition 4.9 (adhesive HLR category)".*

and can be constructed in two phases:

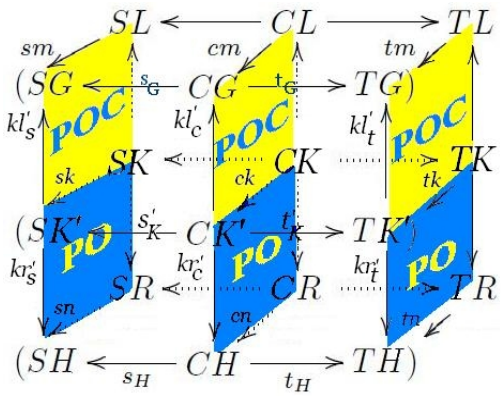


Image 22: ETGT-step: pushouts and pushout complements

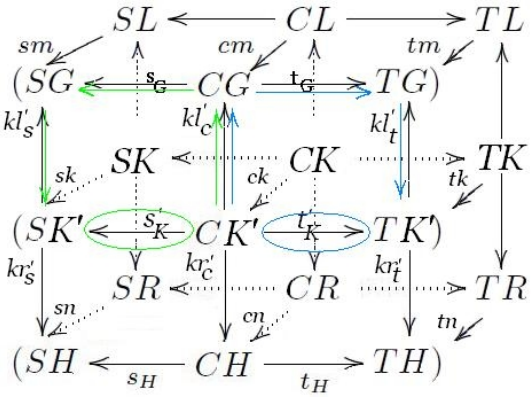


Image 21: ETGT-step: morphism calculation in phase 1

1. *Calculating three pushout complements(definition [EEP+06]"Definition A.20", yellow fields in Image 22):* $SK \xrightarrow{sk} SK' \xrightarrow{kl'_s} SG$, $CK \xrightarrow{ck} CK' \xrightarrow{kl'_c} CG$ *and* $TK \xrightarrow{tk} TK' \xrightarrow{kl'_T} TG$ *with morphisms:* $s_K' := kl_s'^{-1} \circ s_G \circ kl_c'$ *and* $t_K' := kl_t'^{-1} \circ t_G \circ kl_c'$ *(see Image 21).*

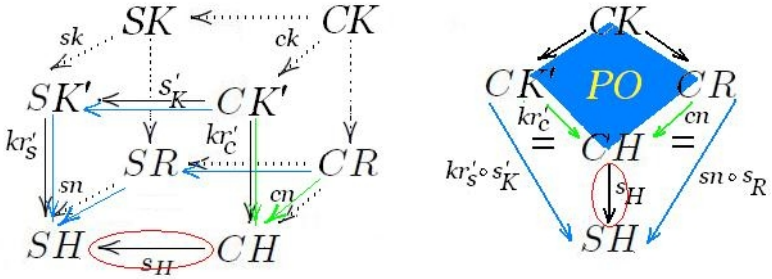


Image 23: Calculation of induced morphism in phase 2 (shown only for s_H*, for* t_H *identically)*

2. *Calculating three pushouts (definition [EEP+06]"Graphs, Typed Graphs, and the Gluing Construction.", blue fields in Image 22):* (SH, kr_s', sn), (CH, kr_c', cn) *and* (TH, kr_t', tn) *with induced morphisms* $s_H := induced(sn \circ s_R, kr_s' \circ s_K', cn, kr_c')$ *and* $t_H := induced(tn \circ t_R, kr_t' \circ t_K', cn, kr_c')$ *(see Image 24).*

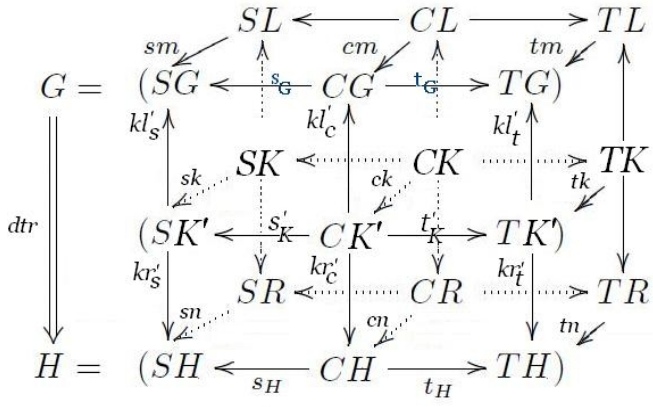


Image 24: DPO Transformation Step

Remark 3 (Condition for component-wise construction):

Pushouts (SG, kl_s', sm), (CG, kl_c', cm), (TG, kl_t', tm) *and* (SH, kr_s', sn), (CH, kr_c', cn) *and* (TH, kr_t', tn) *can be constructed component-wise (with* $SK \xrightarrow{sk} SK' \xrightarrow{kl'_S} SG$, $CK \xrightarrow{ck} CK' \xrightarrow{kl'_C} CG$, $TK \xrightarrow{tk} TK' \xrightarrow{kl'_T} TG$*), only if there are morphisms* s_K' *and* t_K' *with* $s_K' \circ ck = sk \circ s_K$, $kl_s' \circ s_K' = s_G \circ kl_c'$, $t_K' \circ ck = tk \circ t_K$ *and* $kl_t' \circ t_K' = t_G \circ kl_c'$.

Fact 2 (Component-wise construction for ETGT-step for Model Transformation/Integration)

Preconditions from Remark 3 are always valid in ETGT-steps while using triple rules for computation of graph transformation source and forward sequences or integration source-/target- and integration sequences.

1. ***Source sequence****:*

   The source sequences are not created directly. To compute the source sequence, a given source model is parsed using inverse source rules, i.e. rules that are deleting on source component. Each parsing sequence ending at the empty graph specifies a source sequence, which is given by inverse of the parsing sequence.

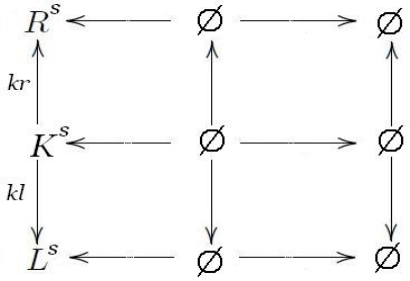


Image 25: Deletion source triple rule

   In this context considered are deleting triple rules $tr_S^{-1}: L_S^{-1} \leftarrow K_S^{-1} \rightarrow R_S^{-1}$ *derived from triple rule* $tr: L \rightarrow R$ *with* $kl = id$.

   The source graph G has also the specific form of $G = (G^S \leftarrow \varnothing \rightarrow \varnothing)$

   $\Rightarrow D = (D^S \xleftarrow{s_K'} \varnothing \xrightarrow{t_K'} \varnothing)$ *and* $s_K' = \varnothing$, $t_K' = \varnothing$.

2. ***Forward sequence****:*

   $tr_F := (L_F \xleftarrow{kl} K_F \xrightarrow{kr} R_F)$, *where* $kl = id$

   $\Rightarrow D = G \Rightarrow (s_K', t_K') = (s_G, t_G)$ *are well defined.*

3. ***Source-/target-*** *and* ***integration sequences*** *can be shown similarly.*

Definition 9 introduces the general idea of a dpo triple rule. However it was not the subject of this thesis to formulate a final dpo rule definition, but to create a dpo rule construct, which under certain conditions could be applied to the triple graph transformation /integration theory. That Definition 9 can be used only under explicit restriction of a gluing condition check can be seen from Image 26.

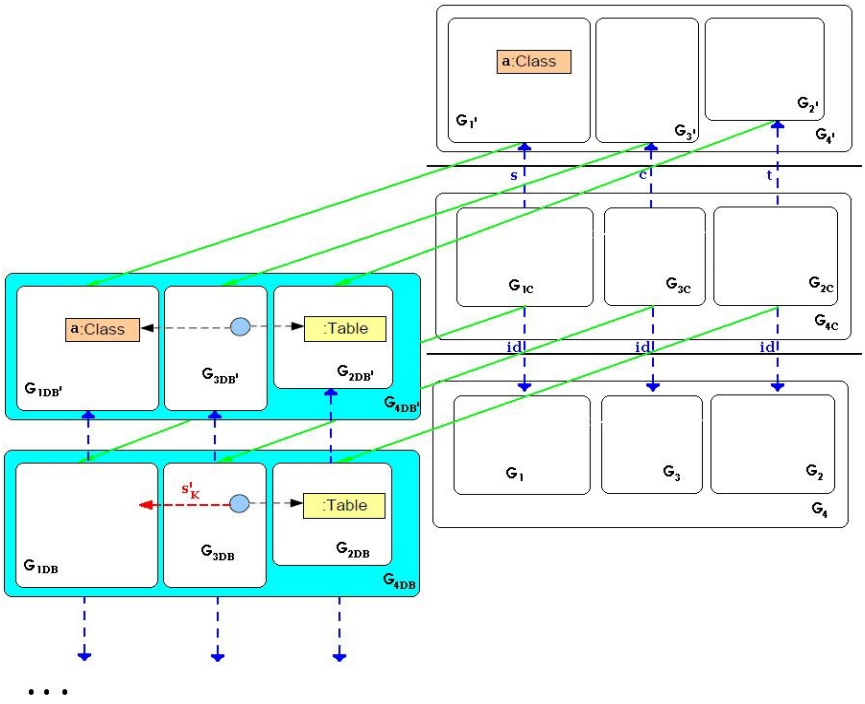


Image 26: Example: ETGT-step fail

In counterexample from Image 26 the triple rule consists of empty graphs except in left-hand side of the rule in source graph $G_{1'}$. The target graph is a simple class2Table graph $G_{4DB'}$ (like in Image 3). The match assigns "a:Class" node in $G_{1'}$ to "a:Class" node in $G_{1DB'}$. During the attempt to apply Definition 9 and perform the ETGT-step, an error occurs: After constructing component-wise three pushout complements $G_{1DB}, G_{2DB}, G_{3DB}$ phase 1 fails, since morphism s_K' is obviously partial.

For still being able to use Definition 9 in the thesis, it was proven in Fact 2 that for specific use of triple graph transformation / integration, conditions from Remark 3 are always fulfilled. So Definition 9 can be used without limitations.

To show the possibility of this approach, to work also in non triple graph transformation / integration environment (yet the final definition of ETGT-step is held open for later works) the example form section 1.2. is taken up and extended by a new rule which is the deletion rule for *Subclass2Table(Image 8) - RestoreSubclass2Table.* It consists of G_4' on left-hand side and G_4 on right-hand. The attempt to construct such a rule with a triple rule according to Definition 3 (Triple Rule tr and Triple Transformation Step) fails, since morphisms s', c', t' become partial instead as expected injective (Image 27).

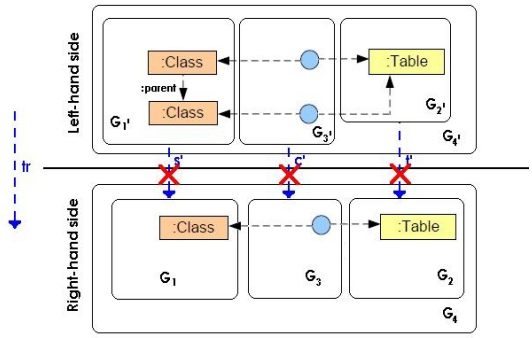


Image 27: Attempt to build up a deletion triple rule according to Definition 3

However, it is possible to build up such a rule using Definition 9 (DPO Triple Rule and Extended Transformation Step). Left-hand and right-hand triple graph stay same as in Image 27, yet the morphisms are replaced by a triple graph structure. For the implementation of triple graph transformation in this thesis, only certain kind of deletion rules are concerned – the ones which are inverse to some given non-deleting triple rules. That for the introduced example as much as the implemented extension of the triple rule in Definition 3 – dpo rule in Definition 9 include some limitations and restrictions.

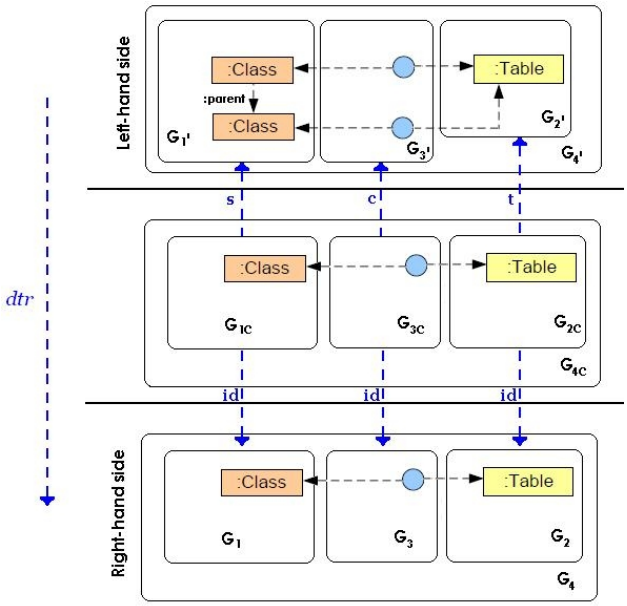


Image 28: DPO triple rule dtr: RestoreSubclass2Table

Therefore *RestoreSubclass2Table* can be constructed as followed: Left-hand and right-hand triple graph are G_4' and G_4 respectively, the partial triple morphism $(s', c', t'): G_4' \to G_4$ is replaced by two injective triple morphisms: $(s, c, t): G_{4C} \to G_4'$ and $(sc, cc, tc): G_{4C} \to G_4$. In order to use all rules defined according to Definition 3 (Triple Rule tr and Triple Transformation Step), one of the triple morphisms is defined as identity triple morphism which is a subset of injective ones (in case of non-deleting rules, it is $(s, c, t): G_{4C} \to G_4'$, in case of deleting rules – $(sc, cc, tc): G_{4C} \to G_4$). This leads to connection graph G_{4C} being an exact copy of either left-hand or right-hand side triple graphs (see Image 28).

Image 29 finally illustrates the application of *RestoreSubclass2Table* to the result triple graph G_{4DB}' from 1.2. . Like in the triple rule application according to Definition 3 to perform this operation a triple graph, a triple rule and a match between them are required. However, to proceed further according to Definition 3, pushouts([EEP+06]"Graphs, Typed Graphs, and the Gluing Construction.") (G_{1DB}, s, sm), (G_{3DB}, c, cm) and (G_{2DB}, t, tm) should be executed, yet this is impossible since morphism s', c', t' are directed opposite the direction of the transformation step. However in Definition 9 Phase 1 a pushout complement([EEP+06]"Definition A.20") is performed. It means that proceeding with the transformation step in this case is to execute pushout complements $G_{1DB'} \xrightarrow{s'} G_{1DB} \xrightarrow{smc} G_{1C}$, $G_{3DB'} \xrightarrow{c'} G_{3DB} \xrightarrow{cmc} G_{3C}$ and $G_{2DB'} \xrightarrow{t'} G_{2DB} \xrightarrow{tmc} G_{2C}$. Since s, c, t are inclusions by definition, so are also s', c', t'. This provides the opportunity to use inverse morphisms $s'^{-1}, c'^{-1}, t'^{-1}$ to calculate $s_{13DB} := s'^{-1} \circ s_{13DB'} \circ c'$ and $t_{32DB} := t'^{-1} \circ t_{32DB'} \circ c'$. The pushout complement result graphs and the calculated morphism combined build a triple graph $G_{4DB} = (G_{1DB} \xleftarrow{s_{13DB}} G_{3DB} \xrightarrow{t_{32DB}} G_{2DB})$. At this point conditions from Remark 3 have to be checked, since in this case they are obviously fulfilled, it can be proceeded to the second phase. To perform the second phase from Definition 9 wont lead to any changes in the result graph, because executing pushouts and calculating induced morphisms from identity morphisms will lead to input triple graph being identical to output triple graph. Therefore phase 2 is skipped. G_{4DB} is the result of applying *RestoreSubclass2Table* to $G_{4DB'}$.

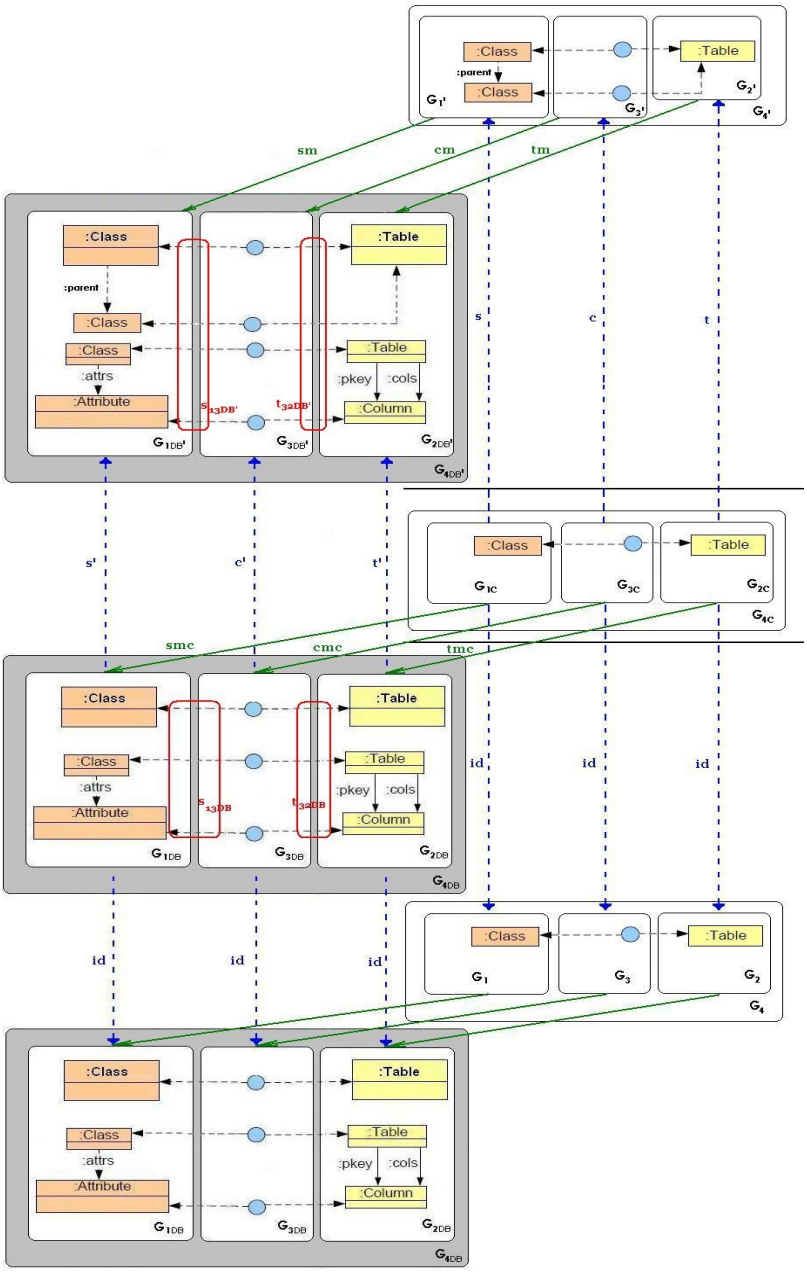


Image 29: Example: Applying a dpo triple rule

Similarly can be shown, how a non-deleting rule (for instance *Subclass2Table*) can be transformed into an extended triple rule and applied on some graph. During non-deleting rule construction morphisms from dpo rule interface triple graph to left-hand side triple graph become identity morphisms. This leads to first phase of Definition 9 being skipped in the application process. But Definition 9 without the first phase converts to Definition 3. So the application of the rule proceeds according to Section 1.2. .

This is the reason for extension of triple rule definition (Definition 9) to have no effect on actual triple graph transformation / integration implementation described in this thesis. For triple graph transformation / integration purposes and as user inputs – non-deleting rules are used with identity morphisms from connection to left-hand side of the rule, so that the extended rule is reduced to triple rule (Definition 3). Yet in transformation- and integration-sequence search the functionality to apply deleting rules is the key for shrinking the search tree and performing an efficiency raise in that way.

III. REALISATION IN WOLFRAM RESEARCH "MATHEMATICA"

1 Wolfram Research "Mathematica"

1.1. Language.

The commercial software Mathematica was developed 1986 by Stephen Wolfram. The software unites different approaches and techniques used also of procedural, object oriented and functional programming. Also Mathematica provides an enormous powerful GUI. With the graphical user interface of Mathematica it is possible to perform any kind of calculation and evaluation. It support a big amount of special symbols and mathematical functions. The possibility to define and visualise in Mathematica any mathematical or physical formula, calculation or evaluation makes interesting to developers and scientists normally using such tools like Matlab.

One of main techniques used in model transformation / integration realization is pattern search. Without it no automated match search(III. 3.9.) can be performed. Without automated match search automated transformation and integration can't be implemented. Yet a pattern search evaluation time grows exponentially with the amount of nodes in the model. Already the model in the second case study needs up to 6 hours evaluation time. For this reason is the possibility of parallel computing, provided by Mathematica of enormous importance for the realisation of model transformation / integration realization. Mathematica supports multitasking and parallel computing which can be realised as automatic parallelization, parallelization of data structures, parallelization for shared memory and synchronisation.

An other reason for choosing this programming language, was the support of functional programming in combination with a powerful visualisation engine.

Functional programming is important, since it creates the possibility to implement the theoretical concepts without converting or extending these concepts with additional information or constructions, i.e. to implement as near to the theory as possible. A realisation in java for instance implies the usage of class/interface structures. But most definitions, theorems and facts in theoretical informatics can be classified as declarations or sets. It is possible to declare and work with sets in most programming language (mostly realised as lists or arrays), but normally sets contain elements of only one type, so even such a simple definition like the definition of a graph II. 1.1. Definition 1, where a graph G is defined by a set $G:=(V,E,s,t)$, isn't easily possible. The reason is that V is a set of nodes, E a set of edge and s and t functions, which means that V and E are of different types. Mathematica doesn't fix the type of the set, so that a graph can be defined directly in the same way it is in the theory: $G:=[V,E,s,t]$ This example shows an other advantage of functional programming. It also provides the possibility to use functions as arguments of methods [Pepper99] "Section 8.1" or use lamda notation [Pepper99] "Section 6.1" during the work with lists.

The visualisation engine of Mathematica provides methods to visualise functions, data, discrete objects, diagrams, images and annotations. It supports professional-quality static and dynamic representations. Methods provided in Mathematica for visual representation of graphs are easy to combine with the theoretical definition of graphs. This options relieve from the necessity to search or develop an additional visualisation engine for graph visualisation.

1.2. Concepts and techniques used in the implementation

In this cheater basic concepts of working with Mathematica are introduced. This is needed for the explanation in the practical part of the thesis (III. 3) to be understandable. For detailed information about work with Mathematica refer to the manual on Wolfram Research homepage([MATH]) or the technical report of Jochen Adamek([Ada09]) ”Cheaters 6.1.-6.5., 7.1.-7.2.”.

Module[*{x, y, ...}, expr*]

Specifies that occurrences of symbols ***x, y, …*** *in* ***expr*** *should be treated as local. Creates new symbols to represent each of its local variables every time it is called. Module can be nested in any way, with inner variables being renamed if necessary.*

In the implementation *Module[]* is used for general programming. Modules appear to be most suitable for the purpose of semi-object oriented programming, since they provide an delimited environment for local variables, which allows to segment the code, similar to method in Java. Mathematica is providing more then only programming possibilities, but also a powerful calculation, evaluation and visualisation engine with a specific user interface in form of notebooks.

Notebook[{ $cell_1$ **,** $cell_2$ **,...}]**

is the low-level construct that represents a notebook manipulated by the Mathematica front end.

Notebooks are saved in files with the ending “.nb” and consists of a defined amount of cells. In each cell is executable code. Each cell can be activated, so that the code is evaluated. A notebook uses standard Mathematica operations, methods and functions, but in case special or user-defined methods are required additional packages can be loaded. Notebooks are not proper for implementing applications, since they provide a real-time evaluation. For programming Mathematica provides *Packages*, which can be included into each other and store executable program code in form of variables, functions and modules. *Packages* are loaded into a notebook with the command *Needs[“...”]*.

Needs[“context`”]

Calls Get["context`"]. By convention, the file loaded in this way is the one which contains a package that defines “context`”. Needs["file`"] typically reads in a file named “file.m”.

Package (.m)

Mathematica source format. Used for storing and exchanging Mathematica programs, packages and data. Plain ASCII text format. Stores Mathematica expressions in InputForm. Can represent program code, numerical and textual data, 2D raster and vector images, 3D geometries, sound, and other kind of data.

As mentioned before, list objects are very common in use with Mathematica and very practical for representing theoretical concepts.

List ({…})

$\{e_{1,}e_{2,}...\}$ is a list of elements. Lists are very general objects, that represent collections of expressions. Nested lists can be used to represent tensors.

Object-like structures like the triple graph or triple rule are realised as lists of different data, which appear normally as the output of a module.

There is a huge amount of functions, which can be applied to lists. Lower are some definitions of the ones used in the description of model transformation practical realisation:

Map[expr]

applies f to each element on the first level in expr.

Select[list,crit]

picks out all elements e_i of list for which crit[e_i] is True.

DeleteDuplicates[list]

deletes all duplicates from list.

Tuples[$list_1$, $list_2$,...]

generates a list of all possible tuples whose i^{th} element is from $list_i$.

Value assignment to a set of variables

A list of variables can be saved in one variable, but the same variable can be assigned to a set of variable (Code 1).

```
In[7]:= a = {1, 2, 3};
        {c, d, e} = a;
        Print["c: ", c]
        Print["d: ", d]
        Print["e: ", e]
c: 1
d: 2
e: 3
```

Code 1

Standard textual graph representation expression (- Graph: <x,y,Directed> -)

Is a standard textual output for a graph object in Mathematica. It represents a summary of general information about components included in the graph: x is the amount of edges, y – the amount of nodes, Directed – explains the graph to have directed edges (default value in this thesis).

The implementation doesn't use this information, since it is obviously incomplete (main data about the connections between edges and nodes is missing), yet it had to be introduced, given that Mathematica Notebook engine(1.1.) uses this graph representation, each time a graph object is in the result set of a calculation, but no explicit graph representation method is specified.

2 Realisation of attributed Graph-transformation in Mathematica(by Jochen Adamek)

2.1. Introduction

In his technical report Jochen Adamek introduces a tool which is a realization of a graph transformation engine for (typed) attributed graphs. In "Konzeption und Implementierung einer Anwendungsumgebung für attributierte Graphtransformation basierend auf Mathematica"([Ada09]) Jochen Adamek explains the basics of (typed) attributed graph transformation theory and the ways of the realization of these concepts by the use of Mathematica. He describes the construction and operating principle of graphs, pushouts, pulbacks and application condition and shows an implementation possibility of these theoretical concepts by the use of Mathematica. He also introduces two benchmarks(Sierpinski and Mutex) applied by the use of his tool.

Jochen Adamek compares his implementation to a tool AGG([AGG]) which was also developed at TU-Berlin. The author compares the advantages, disadvantages and efficiency of techniques, which were implemented in his work, to same techniques realized in AGG. He pays much attention especially to the efficiency analyses and evaluates the efficiency of the application of AGG and Mathematica implementations of the (typed) attributed graph transformation theory to the MUTEX benchmark .

In the section "Zukünftige Realisierungen" the author describes several areas and approaches which might be based on his implementation in future. "Triple graph transformation / integration in Mathematica" is the realization of one of these approaches.

In cheaters 6.1.-6.5., 7.1.-7.2.([Ada09]) the author describes the development environment of Mathematica and Mathematica-Workbench for Eclipse in a very detailed way. The implementation of this thesis' practical part was performed in the same development environment, yet the description of it is limited to Mathematica techniques which have been used. That for, the technical report of Jochen Adamek([Ada09]) is a good reference for the usage of Mathematica and Mathematica-Workbench for Eclipse.

The implementation in this thesis is build up on several methods and concepts developed in [Ada09]. In the following chapter these methods and concepts are briefly introduced.

2.2. Concepts and techniques used in the implementation

Morphisms realisation

Morphisms Jochen Adamek realises as indexed list: the index of an element in it is the index of corresponding node/edge in the morphism source graph node/edge list, the element itself is the index of corresponding node/edge in the morphism target graph node/edge list respectively.

calculateGraphMorphism

Converts a list of node-/edge- pairs into a morphism $f : A \to B$, represented by an indexed List of image positions. Positions of first elements of input tuples in appropriate input graphs become indexes, positions of second elements respectively become elements.

makeRule

calculates a DPO rule for given left hand side LHS with typing morphism typeLHS, right hand side RHS with typing morphism typeRHS and application conditions ACi={kindi, {ACiGraph,typeACi}} with kindi:[\"NAC\",\"PAC\",..]. Result is the complete rule with morphisms.

makeTypedGraph

constructs a graph by the use list of nodes and list of edges. As third argument makeTypedGraph() gets a type graph. Result is a graph with edges and nodes typed over the argument type graph.

pushout

pushout construction for category GRAPHS/TG, assumed that the graph morphism r is injective. Takes three typed graphs and two morphisms as arguments. Returns a pushout object graph and two corresponding morphisms.

inducedGraphPO

calculates the induced mapping h for category SETS. Arguments are morphisms x1, x2, n, g. x1 and x2 are mappings of comparison object X, n and g are mappings of pushout diagram into the pushout object H (set H), i.e. $R \xrightarrow{x1} X \xleftarrow{x2} D$, $R \xrightarrow{n} H \xleftarrow{g} D$. Result induces morphism $h : H \to X$.

compG

calculates the composition of graph morphisms. Gets as arguments two list of nodes of morphisms to combine and two list of edges of morphisms to combine. Output is a morphism.

invG

calculates an inverse of injective graph morphism $f : A \to B$. Arguments are a list of nods and a list of edges mapped by the morphism, amount of nods and edges.

matchesSimple()

calculates all matches from a graph L to a graph G. Arguments are typed graphs L and G, their type graph and inheritance graph.

IsGluingCondition

calculates weather the gluing conditions for a double pushout are fulfilled. Arguments are three graphs and the morphisms between them. Output a boolean value.

3 Realisation of Model Transformation in Mathematica

After formulating theoretical background of the triple graph transformation / -integration, familiarising with Wolfram Research "Mathematica" and bringing in some aspects from realisation of attributed Graph-transformation in "Mathematica" by Adamek Jochen ([Ada09]), the implementation of triple graph transformation / -integration can be finally introduced.

In order of better navigation in program code and packages, first a general model of the software in 3.1. is given. The model concludes: a semi-Class diagram, which specifies the relationships and dependencies between packages and a Use-Case diagram to specify the interfaces for uses input/-output methods implemented.

One of the main objects of this thesis was to construct such an implementation, that it would be as near as possible to the theoretical background. Therefore the present chapter is split in parts according to the theoretical concepts, which are realised. First it is explained how a triple graph and a triple rule are realised and the ways how they can be passed to the application. 3.5. Describes how a triple rule is applied to a triple graph, by executing three pushouts([EEP+06]"Graphs, Typed Graphs, and the Gluing Construction.") on source-, connection-, target- parts of given triple rule and triple graph over three matches, from the left-hand side of the rule to the triple graph. However in the implementation the concept from II. 1.2. Triple Rule is extended by II. 1.5. , so in 3.6.1 its explained, how the realisation of the triple rule and triple rule application were extended to provide deletion functionality to triple rules.

Further the description of source- and target- rules realisation and graph transformation accomplishment follows. Similarly source-/target- and integration- rules generation and graph integration implementation is shown.

The ability to perform a graph transformation / integration looses practical sense if an appropriate match from given graph to some next triple rule in a transformation/integration sequence has to be specified manually in each step. Therefore in 3.9. a solution for this problem is shown. Having a match-search functionality embedded into the transformation/integration methods makes (under the notion of source consistency II. 1.3. ,II. 1.4.) triple graph transformation/integration automated.

3.1. General model of the concept for implementation

It is wrong to speak about classes or objects in Mathematica, given that all objects-like structures are different kinds of sets, or packages, which are just collections of methods, with the possibility to specify weather they are private or public. One cant assign a variable some class's instance, or call methods from some variable. Yet a package is an organisation unit and by including it into an other package – access to additional methods and variables is gained. Moreover some of implemented modules are meant as class-like structures. For instance *TGGmakeGraph* is a method which is implemented as a module for making out of 8 lists and 3 type-graphs a triple graph. The triple graph is represented by 3 graphs, 2 lists (morphisms between the inside graphs), 3 type-graphs and 3 morphisms, which assign source-, connection and target inside graphs to the type graphs. To represent a structure like a triple Graph in object oriented programming language, a variable with class-type like *TripleGraphExample* in Code 2 is be needed. In Mathematica variables have no specific

type before first value assignment. This makes it possible for the value to have any random structure. After assigning a result of *TGGmakeGraph* to some variable, the variable will include all of the components of a triple graph like an instance of *TripleGraphExample* would do.

Based on the observation above, a model for project dependency organisation was created. The intend was to build up a semi-class diagram in UML. The rules for building up a class diagram were replaced with some more appropriate rules for a Mathematica implementation.

Rules for building up a package relationship model:

1. Mathematica packages are represented as classes.
2. Methods used to specify objects are represented as classes.
3. Include-relationships are visualised as directed edges.
4. Include-relationships are organised by colours according to Image 31.
5. Composition-relation is represented by an edge with a filled diamond on the end.
6. The folder structure is visualised as overlapping package-figures in different colours.

```
public class TripleGraphExample {

    Graph sourceGraph;
    Graph sourceGraphTypeGraph;
    Graph connectionGraph;
    Graph connectionGraphTypeGraph;
    Graph targetGraph;
    Graph targetGraphTypeGraph;
    Morphism cs;
    Morphism ct;

    public TripleGraphExample(Graph sourceGraph, Graph sourceGraphTypeGraph,
            Graph connectionGraph, Graph connectionGraphTypeGraph,
            Graph targetGraph, Graph targetGraphTypeGraph, Morphism cs,
            Morphism ct) {
        this.sourceGraph = sourceGraph;
        this.sourceGraphTypeGraph = sourceGraphTypeGraph;
        this.connectionGraph = connectionGraph;
        this.connectionGraphTypeGraph = connectionGraphTypeGraph;
        this.targetGraph = targetGraph;
        this.targetGraphTypeGraph = targetGraphTypeGraph;
        this.cs = cs;
        this.ct = ct;
    }
}
```

Code 2: Possible implementation of a triple graph in OOP

Since Mathematica was used only as an instrument and is not in focus of this thesis, Mathematica packages and dependences are visualised only partially. The TGG (Triple Graph Grammar)(III. 2) is also not wholly visualised but only the parts used by TGT (Triple Graph Transformation).

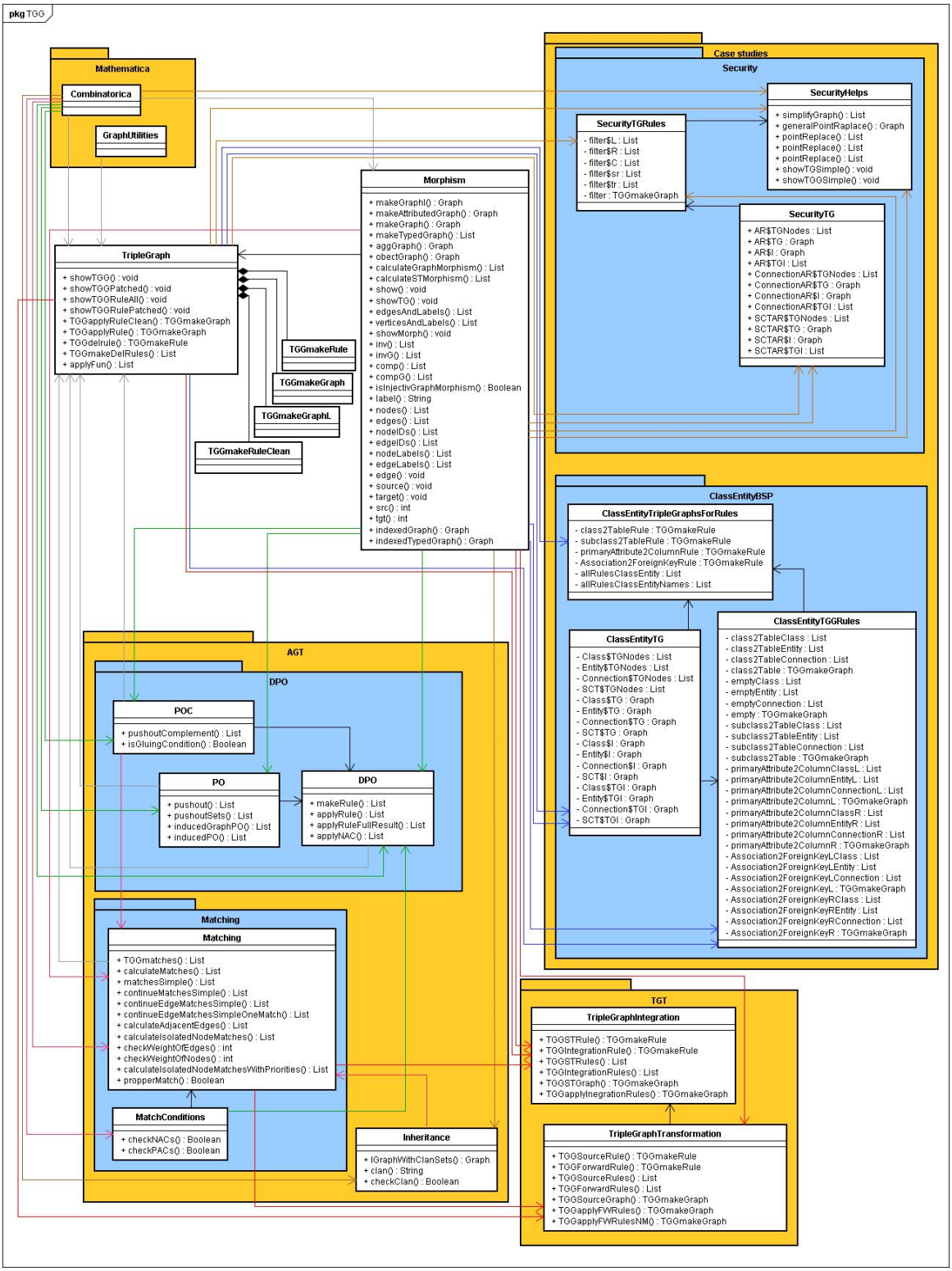


Image 30: Package relationship model

Image 31: Respective colours for include edges

From the Image 30 general organisation of the implementation can be seen. The project is divided in 4 parts (without the Mathematica folder): TGG, AGT, TGT and Case studies.

- TGG (Triple Graph Grammars):

  In the TGG triple graph, triple graph morphism, triple rule and triple graph service methods like graph-drawing methods or rule applying methods are defined and implemented. It is the core of the application since most of other packages use these. The package Morphism is almost wholly adopted from III. 2 Realisation of attributed Graph-transformation in Mathematica(by Jochen Adamek) and used in building up the triple graphs and applying rules.

- AGT (Algebraic Graph Transformation):

  AGT is the III. 2 Realisation of attributed Graph-transformation in Mathematica(by Jochen Adamek) itself, extended by triple graph functionalities. Some packages like RuleApplication.m or TGG.m and all of the examples are excluded from the model since they were not used by current implementation. Some packages like Matching had to be extended.

- TGT (Triple Graph Transformation):

  TGT includes the packages in which the triple graph transformation and integration functionalities are realised. There are methods for generating source- and forward-, sourc-/forward and -integration rules, applying those rules to sequences of triple rules or triple graphs.

- Case studies:

  Case studies include declarations for example applications of model transformation / integration to case studies (real models), described in the end of this thesis IV. .

Generally it is not needed to include a package twice. For example, if the package *Morphism* includes the package *Compbinatorica* and the package *TripleGraph* includes the package *Morphism,* it normally means that the package *TripleGraph* also includes the package *Morphism.* Yet in during implementation complications occurred while using packages from include inheritance. These complications led to the decision to include packages explicitly when they are needed.

The notebooks used for demonstration and evaluation of data are also excluded from the model since those are user-interfaces and can not be included by other packages.

3.2. Use cases

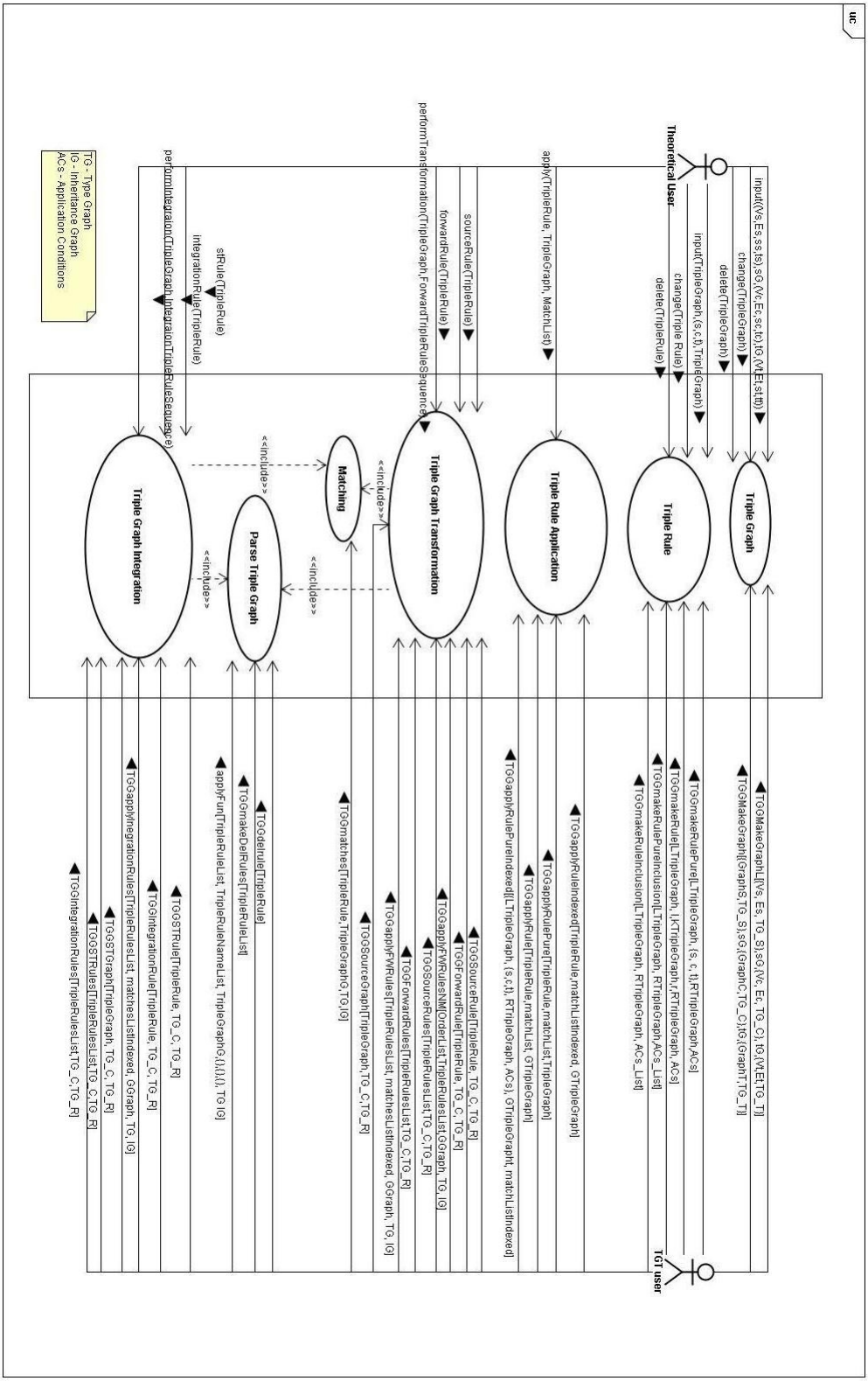


Image 32: TGT use case diagram

The use case diagram in Image 32 consists of two parts. The use cases based on the theoretical concepts from Part II. are visualised in the left part of the diagram. This use cases are the guide lines for the implementation and can be summarised as follows:

- The user can input, edit and delete triple graphs and triple rules.
- The user can apply a triple rule to a triple graph.
- The user can make source and forward rules out of triple rules and execute a graph transformation on a triple graph.
- The user can make source-/target- and integration rules out of triple rules and execute a graph integration on a triple graph.

Aside of the editing and deleting ability which is provided by Mathematica itself, all use cases are realised.

The use cases which are visualised in the right part of the diagram, are the actual implemented methods. They are described step by step in sections 3.3. - 3.10.

The middle part of the diagram represents the interfaces of the application which realize the use cases.

3.3. A Triple Graph as input

```
In[3]:= Needs["ClassEntityTGGRules`"]

In[6]:= class2Table

{
 {-Graph:<0,1,Directed>-, {{1}, {}}},
 {{1}, {}},
 {-Graph:<0,1,Directed>-, {{1}, {}}},
 {{1}, {}},
 {-Graph:<0,1,Directed>-, {{1}, {}}}
}
```

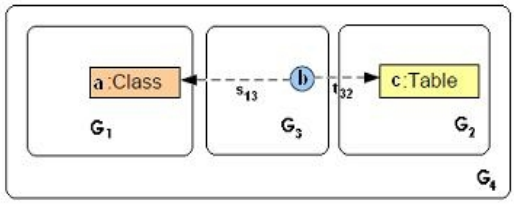


Image 33: Example: class2Table triple graph

The implementation of a triple graph, as mentioned in 3.1. , is realised in the form of a module (III. 1.2.). The triple graph itself is the output of this module in the form of a list. For instance:

The example in Image 33 left-hand side represents a triple graph class2Table in Mathematica. Image 33 Right-hand side is visual representation of the same example from II. 1.1. . As can be seen, calss2Table is a list of 5 elements: 1, 3, 5 elements are again lists which include source-, connection-, target- graphs and their typings; elements 2, 4 are list-realisations of morphisms between those graphs (Code 3).

```
G1 : {-Graph:<0,1,Directed>-, {{1}, {}}}
s13 : {{1}, {}}
G3 : {-Graph:<0,1,Directed>-, {{1}, {}}}
t32 : {{1}, {}}
G2 : {-Graph:<0,1,Directed>-, {{1}, {}}}
G4 : class2Table
or
G4 : {
       {-Graph:<0,1,Directed>-, {{1}, {}}},
       {{1}, {}},
       {-Graph:<0,1,Directed>-, {{1}, {}}},
       {{1}, {}},
       {-Graph:<0,1,Directed>-, {{1}, {}}}
     }
```

Code 3: Example(pseudo code): class2Table fragmentation

There are two ways to assign the application a triple graph as input. In the package *TripleGraph.m,* there are methods *TGGMakeGraph* and *TGGMakeGraphL*. Since the implementation is supposed to be near to theory, but as flexible as possible there are also two approaches to input data.

TGGMakeGraphL

TGGMakeGraphL is used to construct a triple graph. It needs 8 Lists as input and three type-graphs. This method was designed to provide possibility of extending the application by interaction capability with other programming languages or applications. Therefore is the input data is in atomic state(Code 4).

```
Needs["ClassEntityTG`"];
class2Table := TGGmakeGraphL[
   {{{"c1", "ClassClass"}}, {}, Class$TG},
   {{{"ctr1", "c1"}}, {}},
   {{{"ctr1", "ClassTableRel"}}, {}, Connection$TG},
   {{{"ctr1", "t1"}}, {}},
   {{{"t1", "EntityTable"}}, {}, Entity$TG}];
```

Code 4

TGGMakeGraph

TGGMakeGraph is used to construct a triple graph. It uses three graphs as arguments together with their typings and two lists which include nodes and edges connected by the morphisms between connection and source, connection and target graphs respectively. This approach uses the TGG implementation from III. 2 (Code 5).

```
Needs["ClassEntityTG`"];
class2TableClass := makeTypedGraph[{{"c1", "ClassClass"}}, {}, Class$TG];
class2TableEntity := makeTypedGraph[{{"t1", "EntityTable"}}, {}, Entity$TG];
class2TableConnection := makeTypedGraph[{{"ctr1", "ClassTableRel"}}, {}, Connection$TG];
class2Table := TGGmakeGraph[class2TableClass, {{{"ctr1", "c1"}}, {}},|class2TableConnection, {{{"ctr1", "t1"}}, {}},
  class2TableEntity];
```

Code 5

As shown in Image 33 both methods return a list containing the graphs and morphisms between them. The only difference is that *TGGMakeGraph* just calculates the morphisms out of lists with node-/edges mappings and returns them together with the input graphs, but *TGGMakeGraphL* first constructs the source-, connection- and target- graphs out of nodes, edges and type graphs, and calls *TGGMakeGraph* afterwards.

3.4. A Triple Rule as input

As it was explained in II. 1.2. , a triple rule is represented by two triple graphs and three morphisms between them. The realisation of this concept is similar to the one of triple graph.

TGGmakeRulePure

TGGmakeRulePure is used to construct a triple rule. The method TGGmakeRulePure takes two triple graphs, three lists of node/edge pairs – morphisms in rough form and a list of application conditions as arguments. The triple graphs are returned without being changed, but the pair lists and corresponding graphs are forwarded to makeRulePure which generates with this information an indexed list for morphism representation (see III. 2.2.).

```
In[62]:= Needs["ClassEntityTG`"];
         Needs["ClassEntityTGGRules`"];
         class2Table := TGGmakeGraph[class2TableClass,
            {{{"ctr1", "c1"}}, {}}, class2TableConnection,
            {{{"ctr1", "t1"}}, {}}, class2TableEntity];
         subclass2Table := TGGmakeGraph[subclass2TableClass,
            {{{"ctr1", "c1"}, {"ctr2", "c2"}}, {}},
            subclass2TableConnection,
            {{{"ctr1", "t1"}, {"ctr2", "t1"}}, {}}, subclass2TableEntity];
         test2 = TGGmakeRulePure [
           class2Table,
           {{{"c1", "c1"}}, {} }, {{{"ctr1", "ctr1"}}, {}},
            {{{"t1", "t1"}}, {}}},
           subclass2Table, {}]
Out[66]= {{{-Graph:<0,1,Directed>-, {{1}, {}}},
           {{1}, {}}, {-Graph:<0,1,Directed>-, {{1}, {}}},
           {{1}, {}}, {-Graph:<0,1,Directed>-, {{1}, {}}}},
          {{{1}, {}}, {{1}, {}}, {{1}, {}}},
          {{-Graph:<1,2,Directed>-, {{1, 1}, {6}}},
           {{1, 2}, {}}, {-Graph:<0,2,Directed>-, {{1, 1}, {}}},
           {{1, 1}, {}}, {-Graph:<0,1,Directed>-, {{1}, {}}}}, {}}
```

Code 6

An example of for *TGGmakeRulePure* execution can be seen in Code 6. The Variables *class2Table* and *subclass2Table* are first initialized with left-hand side triple graph and right-hand side triple graph respectively. Then these varaibles are passed to *TGGmakeRulePure* as arguments. The content of *class2Table* is green underlined in the output of *TGGmakeRulePure*. After comparing it to G_4 in Code 3, it becomes obvious that *TGGmakeRulePure* does not perform any manipulation with the graphs themselves. Yet the second argument (blue underlined in the input and brown in the output Code 3) is transformed from a list of lists of name tuple's to a list of lists of indexes. These lists of indexes represent the morphisms between the left- and right- hand side source-, connection- and target graphs of the triple rule.

The purple marked empty list in the input and output parts of *TGGmakeRulePure* is the list of application conditions. The extension of model transformation / integration by application conditions is left open for future works (V. 1.2.).

3.5. Applying a Triple Rule on a Triple Graph

In II. 1.2. a triple graph transformation step is described. That procedure is needed to apply any triple rule on a triple graph. The required inputs are a triple rule, a triple graph and a proper match from the given triple rule to the given triple graph. The result is a triple graph, obtained by applying the given rule to the given graph with the given match. This means that elements in the given graph, which correspond to then given rule's left hand side elements are removed and the elements from the right hand side are created instead. A method from the implementation realising pictured procedure is TGGapplyRulePure.

TGGapplyRulePure

TGGapplyRulePure is used to apply a triple rule on a triple graph. TGGapplyRulePure takes as arguments: a triple rule (output of TGGmakeRulePure(3.4.)), a triple graph (output of TGGMakeGraph or TGGMakeGraphL(3.3.)) and a match from left-hand side of the triple rule to given triple graph.

```
In[50]:= Needs["ClassEntityTG`"];
         Needs["ClassEntityTGGRules`"];

         empty := TGGmakeGraph[emptyClass, {{}, {}}, emptyConnection,
            {{}, {}}, emptyEntity];
         class2TableRulePure :=
           TGGmakeRulePure [empty, {{ {}, {} }, { {}, {}}, { {}, {}}},
            class2Table, {}];
         test = TGGapplyRulePure [class2TableRulePure , empty,
           {{{}, {}}, {{}, {}}, {{}, {}}}]

Out[54]= {{-Graph:<0,1,Directed>-, {{1}, {}}},
          {{1}, {}}, {-Graph:<0,1,Directed>-, {{1}, {}}},
          {{1}, {}}, {-Graph:<0,1,Directed>-, {{1}, {}}}}
```

Code 7: Applying class2TableRulePure to an empty triple Graph

Code 7 represents a simple case of *TGGapplyRulePure* execution: blue underlined is the rule to apply, green – the host graph, red – the match; the output (Out[54]) is a triple graph in form of a list like in Section 3.3. .

To describe the different steps of *TGGapplyRulePure* implementation the example used in II. 1.2. appears to be more detailed, so more suitable.

```
In[12]:= Needs["ClassEntityTG`"];
         Needs["ClassEntityTripleGraphsForPureRules`"];
         G1 DB := makeTypedGraph[
            {{"c1", "ClassClass"}, {"c2", "ClassClass"},
             {"at1", "ClassAttribute"}},
            {{"att1", {"c2", "at1"}, "Class$attrs"}}, Class$TG];
         G2 DB := makeTypedGraph[
            {{"t1", "EntityTable"}, {"t2", "EntityTable"},
             {"cl1", "EntityColumn"}},
            {{"pk1", {"t2", "cl1"}, "Entity$pkey"},
             {"cl1", {"t2", "cl1"}, "Entity$cols"}}, Entity$TG];
         G3 DB := makeTypedGraph[
            {{"ctr1", "ClassTableRel"}, {"ctr2", "ClassTableRel"},
             {"acr1", "AttrColRel"}}, {}, Connection$TG];
         G4 DB := TGGmakeGraph[G1 DB,
            {{{"ctr1", "c1"}, {"ctr2", "c2"}, {"acr1", "at1"}}, {}},
            G3 DB, {{{"ctr1", "t1"}, {"ctr2", "t2"}, {"acr1", "cl1"}},
            G2 DB];
         subclass2TableRulePure :=
           TGGmakeRulePure [class2Table,
            {{{{"c1", "c1"}}, {}}, {{{"ctr1", "ctr1"}}, {}},
             {{{"t1", "t1"}}, {}}}, subclass2Table, {}];
         G4 DB' = TGGapplyRulePure [subclass2TableRulePure , G4 DB,
            {{{{"c1", "c1"}}, {} }, {{{"ctr1", "ctr1"}}, {}},
             {{{"t1", "t1"}}, {}}}]
Out[18]= {{-Graph:<2,4,Directed>-, {{1, 1, 3, 1}, {1, 6}}},
          {{1, 2, 3, 4}, {}}, {-Graph:<0,4,Directed>-,
           {{1, 1, 3, 1}, {}}}, {{1, 2, 3, 1}, {}},
          {-Graph:<2,3,Directed>-, {{1, 1, 2}, {3, 4}}}}
```

Code 8: TGGApplyRulePure

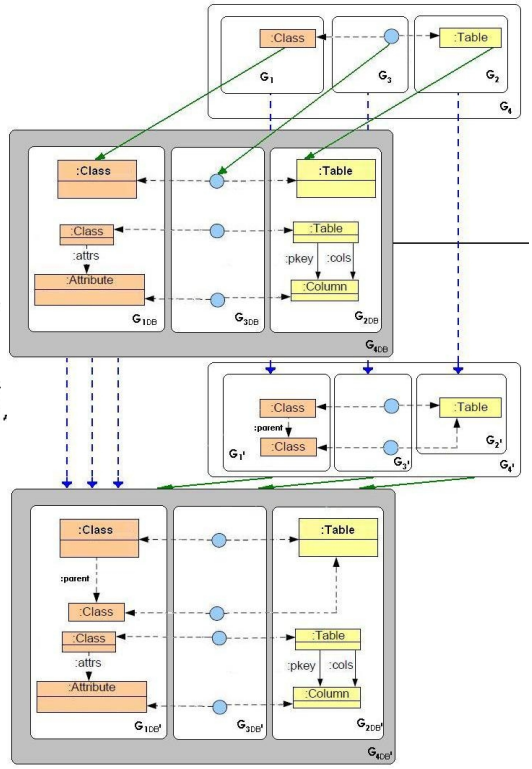


Image 34: Applying a triple rule to a triple graph

In Code 8 *subclass2TableRulePure* is applied to the example graph G_{4DB} from Image 34 with a triple graph $G_{4DB'}$ as resulting output(Out[18]). *TGGApplyRulePure* takes mentioned above arguments and converts the list of match pairs into a list of indexed list morphism representations (III. 2.2.) by applying *calculateGraphMorphism*(III. 2.2.) to match-pairs with corresponding graphs from left-hand side of the rule and the input graph (Code 9 blue underlined).

```
msGMorphism=calculateGraphMorphism[ms,LS[[1]],GS[[1]]];
mcGMorphism=calculateGraphMorphism[mc,LC[[1]],GC[[1]]];
mtGMorphism=calculateGraphMorphism[mt,LT[[1]],GT[[1]]];
TGGapplyRulePureIndexed [tripleRule,
    G, {msGMorphism,mcGMorphism,mtGMorphism}]
```

Code 9

Code 10 shows the results of these calculations in case of application to the example.

```
{{1}, {}} = calculateGraphMorphism[{{{"c1", "c1"}}, {} },
   class2Table[[1]][[1]], G1DB];
{{1}, {}} = calculateGraphMorphism[{ {{"ctr1", "ctr1"}}, {}},
   class2Table[[3]][[1]], G3DB];
{{1}, {}} = calculateGraphMorphism[{ {{"t1", "t1"}}, {}},
   class2Table[[5]][[1]], G2DB];
TGGapplyRulePureIndexed [subclass2TableRulePure , G4DB,
 {{{1}, {}}, {{1}, {}}, {{1}, {}}}]
```

Code 10: Example (pseudo code)

Further, the generated lists and most input information are passed to *TGGapplyRulePureIndexed* (Code 9 green underlined) which performs the actual rule application. In TGGapplyRulePureIndexed the input data is first split into source-, connection- and target parts (Code 11).

```
{LS, sL, LC, tL, LT} = L;
{RS, sR, RC, tR, RT} = R;
{GS, sg, GC, tg, GT} = G;
{s, c, t} = sct;
{ms, mc, mt} = msct;
```

Code 11

Code 12 has a demonstrative purpose. The variables from Code 11 are replaced with their values from the example. During the real execution, each code line is a value assignment. For instance the variable *LS* (Code 11) gets the value of the first element from the list *class2Table* (which is G_1), the variable *sL* gets the value of the second element of class2Table, etc... (see also III. 1.2. "Value assignment to a set of variables")

```
{G1, {{1}, {}}, G3, {{1}, {}}, G2} = class2Table;
{G1', {{1, 2}, {}}, G3', {{1, 1}, {}}, G2'} = subclass2Table;
{G1DB, {{1, 2, 3}, {}}, G3DB, {{1, 2, 3}, {}}, G2DB} = G4DB;
{s, c, t} = {{{1}, {}}, {{1}, {}}, {{1}, {}}};
{ms, mc, mt} = {{{1}, {}}, {{1}, {}}, {{1}, {}}};
```

Code 12: Example (pseudo code)

The assigned source-, connection- and target values are passed to the method *pushout* (see III.2.2) (Code 13).

```
{{HS, typeHS}, sn, s1} = pushout[LS, s, RS, ms, GS];
{{HC, typeHC}, cn, c1} = pushout[LC, c, RC, mc, GC];
{{HT, typeHT}, tn, t1} = pushout[LT, t, RT, mt, GT];
```

Code 13

```
{{G1DB', typeHS}, sn, s1} = pushout[G1, {{1}, {}}, G1', {{1}, {}}, G1DB];
{{G3DB', typeHC}, cn, c1} = pushout[G3, {{1}, {}}, G3', {{1}, {}}, G3DB];
{{G2DB', typeHT}, tn, t1} = pushout[G2, {{1}, {}}, G2', {{1}, {}}, G2DB];
```

Code 14: Example (pseudo code)

Outputs of pushout applications are lists (containing source-, connection- and target result graphs and morphisms) which are used as arguments in *inducedGraphPO*(III.2.2) and *compG*(III.2.2). By the use of *inducedGraphPO()* and *compG()* induced morphisms are generated which connect connection- and source- / connection- and target- graphs of the output triple graph (Code 15, Code 16).

```
sh = inducedGraphPO[compG[sn, sR], compG[s1, sg], cn, c1];
th = inducedGraphPO[compG[tn, tR], compG[t1, tg], cn, c1]
```

Code 15

```
sh = inducedGraphPO[compG[{{1, 4}, {2}}, {{1, 2}, {}}],
   compG[{{1, 2, 3}, {1}}, {{1, 2, 3}, {}}], {{1, 4}, {}}, {{1, 2, 3}, {}}];
th = inducedGraphPO[compG[{{1}, {}}, {{1, 1}, {}}],
   compG[{{1, 2, 3}, {1, 2}}, {{1, 2, 3}, {}}], {{1, 4}, {}},
   {{1, 2, 3}, {}}];
```

Code 16: Example (pseudo code)

In the end, the outputs of all methods are summarised in the form of a triple graph (output of TGGMakeGraph or TGGMakeGraphL(3.2)) and returned (Code 17).

```
{{HS, typeHS}, sh, {HC, typeHC}, th, {HT, typeHT}}
```

Code 17

If replaced $G_{1DB'}, G_{3DB'}$ and $G_{2DB'}$ in Code 18 by the textual standard graph representation expression (1.2.), the result would be the output(Out[18]) from Code 8.

```
{{G1 DB', {{1, 1, 3, 1}, {1, 6}}}, {{1, 2, 3, 4}, {}},
 {G3 DB', {{1, 1, 3, 1}, {}}}, {{1, 2, 3, 1}, {}}, {G2 DB', {{1, 1, 2}, {3, 4}}}}
```

Code 18: Example (pseudo code)

3.6. Triple Rule modification

During the working process, it became obvious that both theoretical and practical approaches need to be extended. On one hand to simplify the work with the resulting application itself, on the other hand to shrink enormous search trees, which are created in triple graph transformation / integration implementation automated application and match search.

3.6.1 DPO rule extension

One of the major modifications is the in chapter II. 1.5. explained DPO triple rule approach which extends triple rule from Definition 3 (Triple Rule tr and Triple Transformation Step) and replaces it in the implementation.

The decision about the implementation of the DPO triple rule had to be taken in early stages of work, because the triple rule is one of the main tools used in building up both – triple graph transformation and integration sequences. Despite a half implemented alternative version the choice was made in favour of the DPO rule. The reason was that the usage of DPO rule concept opens the way for deletion rule generation. While deletion rules give way to a backward pattern search implementation. Normally the pattern search is performed brute-force by trying all available rules from the input set. An algorithms based on this approach stops, when in one of the constructed search tree leafs the aimed graph occurs. A backward pattern search starts at the aimed graph and proceeds by applying the corresponding deletion rules from available input set to the graph. The algorithms based on this approach stops at the empty graph. If no rule can be applied any more, but the empty graph is not reached, backtracking is performed (the transformation / integration sequence search algorithms is described in chapter 3.10.). This approach appeared to be far more productive then the brute-force search, since it shrinks the search tree enormously.

To extend the implementation by DPO rule functionality, first, *makeRulePure (3.4.)* had to be reconstructed into *makeRule.* Its functionality stayed same but the input and output data changed. The method *makeRule* becomes three graphs and two pair lists (which represent morphisms between these graphs) as input, converts pair lists to indexed lists morphism representation (III. 2.2.) and returns a triple graph.

Correspondingly to the method *makeRulePure, the method TGGmakeRulePure is* converted into *TGGmakeRule.* Changes are similar to those made in *makeRulePure*: the input data is extended by an additional triple graph and a triple morphism list.

TGGmakeRule

TGGmakeRule is used to construct a DPO triple rule. The method TGGmakeRule takes three triple graphs, six lists of node/edge pairs – morphisms in rough form and a list of application conditions as arguments. The triple graphs are returned without any change, but the pair lists and corresponding graphs are forwarded to makeRule, which generates with this information an indexed list for morphism representation (see III. 2.2.). The output is a DPO rule with application conditions.

Changes made in the method *TGGapplyRulePure* are bigger. First step from *TGGapplyRule* to *TGGapplyRuleIndexed* stays same. *TGGapplyRule* just converts the input morphism tuples into indexed lists and passes all arguments further. Yet in *TGGapplyRuleIndexed* it is not enough to calculate three pushouts and two induced morphisms any more, since the arguments for performing this action are simply missing. According to Definition 9 (DPO Triple Rule and Extended Transformation Step) to proceed to phase 2 (pushout and induced), firstly three pushout complement and two morphisms from phase 1 have to be determined.

```
{ {DS, kS, fS},{HS, nS, gS} }=applyRuleFullResult[{LS,lS,KS,rS,RS,{}},mS,GS];
{ {DC, kC, fC},{HC, nC, gC} }=applyRuleFullResult[{LC,lC,KC,rC,RC,{}},mC,GC];
{ {DT, kT, fT},{HT, nT, gT} }=applyRuleFullResult[{LT,lT,KT,rT,RT,{}},mT,GT];
```

Code 19: combined POC and PO calculation

For this reason the input graph, matches and DPO rule left hand side after being spit in source- (*GS, mS, {LS,lS,KS,rS,RS}*), connection- (*GC, mC, {LC,lC,KC,rC,RC}*) and target- (*GT, mT, {LT,lT,KT,rT,RT}*) parts are passed to *applyRuleFullResult* (see Code 19).

applyRuleFullResult

applyRuleFullResult is the realisation of a transformation ([EEP+06]"Definition 5.2 (transformation)") application. The method applyRuleFullResult takes as arguments: a production in form of three graphs and morphisms between them, a typed graph and a match – in form of an indexed list morphism (III. 2.2.). The method checks the match to be consistent with the gluing conditions([EEP+06]"Definition 6.3 (gluing condition in adhesive HLR systems)"), constructs the pushout complement and the pushout in the second phase of the transformation. The output contains the pushout complement graph with appropriate morphisms and a pushout graph with appropriate morphisms.

The application of *appyRuleFullResult* is based on the observation, that the pushout complements / pushouts can be calculated separately from morphisms, connecting source-, connection- and target- parts of a DPO triple rule (under the notion of Definition 8 Fact 2) and that combination of pushout complement followed by a pushout is a transformation.

```
sD=compG[invG[fS,GSamounts],compG[sG,fC]];
tD=compG[invG[fT,GTamounts],compG[tG,fC]];
```

Code 20: Calculation of connection morphisms

Morphisms calculated with pushout complements(*fS, fC, fT*) are passed to *compG(...)* und *invG(...)* (see III. 2.2.) in order to compute connection morphisms *sD* and *tD* according to the formulas $s_D' := fS^{-1} \circ sG \circ fC$ and $tD := fT^{-1} \circ tG \circ fC$ from Definition 9 phase 1(Code 20).

After all components of the connection graph from an ETGT-step(Definition 9) are determined *TGGapplyRuleIndexed* proceeds same way as *TGGapplyRulePureIndexed by* calculating induced morphisms for the output triple graph (Code 21) since the creation of source-, connection- and target- graph parts of it by the use of pushouts has been already performed in *appyRuleFullResult.*

```
sH=inducedGraphPO[compG[nS,sR],compG[gS,sD],nC,gC];
tH=inducedGraphPO[compG[nT,tR],compG[gT,tD],nC,gC];
```

Code 21: TGGapplyRuleIndexed induced Calculation

The method *TGGapplyRuleIndexed* returns same way as *TGGapplyRulePureIndexed* a triple graph (output of TGGMakeGraph or TGGMakeGraphL(3.2)).

3.6.2 Inclusion extension

Yet the extension of triple rule implementation by dpo rule functionality has let to an other problem. The application implementation in this thesis is the realisation of graph transformation theory based on the triple graph approach, which implies the use cases for this application to be fixed (see 3.2.). As can be seen from diagram in Image 32 the aimed final user of this application will create simple triple rules according to Definition 3 and perform a graph transformation / integration. This user does not know about the existence of a dpo triple rule concept (Definition 9) and about the positive influence of this concept on the efficiency of transformation / integration sequence calculation. However he ofcourse welcomes an efficiently and fast working application. This makes the task specification obvious: implemented extension of the triple rule must be reduced to the input schema of given use case (3.2.) without loosing its functionality. Simplified it means that the user should think himself to be in a world without chapter II. 1.5. , yet the application should use dpo rules for transformation / integration sequences calculations.

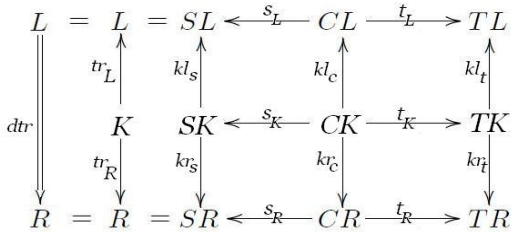


Image 36: DPO triple rule

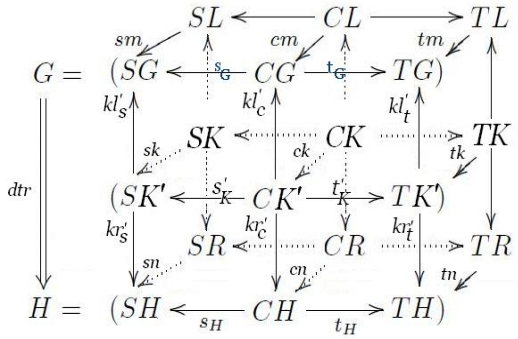


Image 35: ETGT-step

The approach, which is used to solve this problem was already mentioned once in Section II. 1.5. . On the one hand is the interest for DPO rules in this thesis limited with deletion rules, on the other hand are deletion rules of interest only to the application itself, while performing transformation / integration sequence calculations. This means – during user input and creation dpo rules look like simple triple rules, yet while being applied an ETGT-step(Definition 9) is performed. From dpo rule definition it can be seen that if phase 1 of an ETGT-step is skipped Definition 9 converts to Definition 3, which leads input data to shrink till the requirements of use case.

Skipping the ETGT-step phase 1 is same as applying ETGT-step phase 1 with no effect – no changes from G to K' (Image 35). This is possible if kl_s, kl_c, kl_t (Image 36) are identity morphisms.

Based on the idea mentioned above *TGGmakeRuleInclusion()* was implemented.

TGGmakeRuleInclusion

Is used to construct a DPO triple rule with input data from TGGmakeRulePure. The method TGGmakeRuleInclusion takes two triple graphs, three lists of node/edge pairs – morphisms in rough form and a list of application conditions as arguments. An interface triple graph is generated, which is the intersection of left-hand source- and right-hand source-, left-hand connection- and right-hand connection, left-hand target- and right-hand target graphs. Morphisms converting the interface triple graph into left-hand and respectively right-hand side triple garph are calculated. Further these morphisms are converted into indexed list for morphism representation (see III. 2.2.). The output is a DPO rule with application conditions.

TGGmakeRuleInclusion() splits the input data into left-hand source, connection and target graphs with morphisms between them, and right-hand respectively(Code 22) and passes the graphs grouped source-source, connection-connection, target-target to *makeRuleInclusion()(Code 23).*

```
{LS, sL, LC, tL, LT} = L;
{RS, sR, RC, tR, RT} = R;
```

Code 22

```
{any1, lS, KS, rS,any2, completeSourceACs} = makeRuleInclusion[LS, RS, sourceACs];
{any1, lC, KC, rC, any2, completeCorrespondenceACs} = makeRuleInclusion[LC, RC, correspondenceACs];
{any1, lT, KT, rT,any2, completeTargetACs} = makeRuleInclusion[LT, RT, targetACs];
```

Code 23

makeRuleInclusion

a) Generates a new Graph(Code 24).

```
(* initially empty graph *)
 {K, typeK} = makeTypedGraph[{}, {}, AR$TG];
```

Code 24

b) Calculates the common edges and nodes between the input Graphs(Code 25).

```
keptNodePositions=commomElementPositions[nodeLabels[L],nodeLabels[R]];
nodesK = nodes[L][[keptNodePositions]]; (*nodes*)
keptEdgePositions=commomElementPositions[edgeLabels[L],edgeLabels[R]];
edgesK = edges[L][[keptEdgePositions]]; (*edges*)
```

Code 25

c) Calculates the typing for the new Graph and inclusion morphisms from new Graph to input Graphs (Code 26).

```
(*typing*)
typeKV = Map[(typeLV[[Position[nodeLabelsL,#][[1]][[1]] ]]) &, nodeLabelsK];
typeKE = Map[(typeLE[[Position[edgeLabelsL,#][[1]][[1]] ]]) &, edgeLabelsK];
typeK = {typeKV, typeKE};
(*calculate inclusion morphisms*)
lV = Map[({#, #}) &, nodeLabelsK]; (*node pairs*)
lE = Map[({#, #}) &, edgeLabelsK];(*edge pairs*)
rV=lV;
rE=lE;
```

Code 26

d) inserts the common edges into the new graph(a)) (Code 27).

```
K[[1]] = edgesK;
K[[2]] = nodesK;
```

Code 27

e) Calculates the production([EEP+06] "Definition 5.1 (production)") using two input and generated in a) graphs (Code 28).

```
resR=makeRule[{L, typeL},{lV,lE},{K,typeK},{rV,rE}, {R, typeR}, ACs];
```

Code 28

In other words *makeRuleInclusion()* calculates the intersection between a pair of graphs and returns a production $p=(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ ([EEP+06] "Definition 3.1 (graph production)"), where L and R are the input graphs. In this way $SL \overset{kl_s}{\leftarrow} SK \overset{kr_s}{\rightarrow} SR$, $CL \overset{kl_c}{\leftarrow} CK \overset{kr_c}{\rightarrow} CR$ and $TL \overset{kl_t}{\leftarrow} TK \overset{kr_t}{\rightarrow} TR$ (Image 36) are established. Next – *TGGmakeRuleInclusion()* constructs $s_K = kl_c \circ s_L \circ kl_s^{-1}$ and $t_K = kl_c \circ t_L \circ kl_t^{-1}$ (Code 29), summarises the application conditions and returns the dpo rule (Code 30).

```
sK = compG[invG[lS,LSamounts], compG[sL, lC]];
tK = compG[invG[lT,LTamounts], compG[tL, lC]];
```

Code 29

```
rule = {L, l, K, r, R, completeACs}]
```

Code 30

Therefore often *makeRuleInclusion()* produces an copy of the left-hand side triple graph. This happens in the case the intersections between left-hand graphs and right-hand graphs result in the left-hand graphs, i.e. left-hand triple graph is a sub-graph of the right-hand one. It is the usual case, yet not the rule. In case of deleting rules it is the way around: the interface triple graph becomes a copy of the right-hand triple graph.

3.7. Model Transformation Realisation

Model transformation realisation consists of several methods: *TGGSourceRule(), TGGForwardRule(), TGGSourceRules(), TGGForwardRules(), TGGSourceGraph(), TGGapplyFWRules() and TGGapplyFWRulesNM()*.

TGGSourceRule() and *TGGForwardRule()* are practical realisations of source and forward rules construction (as explained in II. 1.3.).

TGGSourceRule

Takes dpo rule (3.6.1) and a list of application conditions as argument, converts the dpo rule into a source rule and returns the result in form of a dpo rule(3.6.1) .

```
sl = {{}, {}}; sr = {{}, {}};
tl = {{}, {}}; tr = {{}, {}};
LC = makeTypedGraph[{}, {}, TGC];
RC = makeTypedGraph[{}, {}, TGC];
LT = makeTypedGraph[{}, {}, TGR];
RT = makeTypedGraph[{}, {}, TGR];
LG = TGGmakeGraph[LS, sl, LC, tl, LT];
RG = TGGmakeGraph[RS, sr, RC, tr, RT];
rule = TGGmakeRuleInclusion[LG, RG, completeACs]
```

Code 31: source rule construction

During source rule construction two new triple graphs are generated out of the source components of triple graphs from the input rule, new generated empty graphs and empty morphisms. This data is afterwords passed to *TGGmakeRuleInclusion*(), so that resulting output is a dpo rule (Code 31).

TGGForwardRule

Takes dpo rule (3.6.1) and a list of application conditions as argument, converts the dpo rule into a forward rules and returns the result in form of a dpo rule(3.6.1).

```
{LS, sl, LC, tl, LT} = L;
{RS, sr, RC, tr, RT} = R;
sl = compG[r[[1]], sl];
LG = {RS, sl, LC, tl, LT};
rule = TGGmakeRuleInclusion[LG, R, completeACs];
```

Code 32: forward rule construction

TGGForwardRule() replaces the left-hand side source graph with right hand side source graph and the morphism transforming the connection graph to source graph in left-hand triple graph with the concatenation of this morphism with the original morphism connecting the source graph of the interface triple graph with the source graph of right hand one (as it can be noticed, in *TGGForwardRule* it is assumed that the left-hand triple graph and the interface graph are equal, i.e. that the input rule is a simple non-deleting rule). The identity morphisms and new interface triple graph are generated afterwords in *TGGmakeRuleInclusion*() (Code 32).

```
TGGSourceRules[rulesList_List, TGC_, TGR_] :=
    Module[{}, Map[TGGSourceRule[#1, TGC, TGR] &, rulesList]];

TGGForwardRules[rulesList_List, TGC_, TGR_] :=
    Module[{}, Map[TGGForwardRule[#1, TGC, TGR] &, rulesList]];
```

Code 33

TGGSourceRules() and *TGGForwardRules()* are methods for applying *TGG-SourceRule()* and *TGGForwardRule()* respectively to lists of triple rules. The implementation of this methods shows the advantage of using Mathematica as programming language: in object oriented language common approach for applying a method to a list of user-defined objects (which triple rules are) is iteration over the list by the use of a "for" or "while" loops. Loops make the program-code heavy and raise the computation time. In Mathematica – this methods can be realised by a mapping function applied to a list, since this programming language supports passing methods as arguments (III. 1.1.), working with list-elements independently of their typing and the "Wild-card"-notation. (III. 1.1.).

In Code 33 the argument list "rulesList" from *TGGSourceRules*() is passed to the mapping function. The mapping function by the use of a "Wild-card" symbol "#1" applies *TGGSourceRule*() to every element in the list and returns a list as a result. *TGGForward-Rules()* is build up in same way.

TGGSourceGraph() generates a source graph out of a triple graph. The method takes a triple graph(output of *TGGMakeGraph* or *TGGMakeGraphL*(3.2)) as input, extracts the source graph, generates new empty graphs, new empty morphisms and passes this arguments to *TGGmakeGraph()*. The output is a triple graph.

TGGapplyFWRules

becomes a set of forward rules(output of TGGForwardRules()), a set of source matches, a source triple graph(output of TGGSourceGraph()), a type graph and an inheritance graph as input. The method applies the forward rules from the input sequence to the input source graph. The result returned by the method is a triple graph and a boolean value, which states weather the application procedure was performed successfully.

TGGapplyFWRules() is a recursive method, which applies a set of rules to a source triple graph. Working with this method, one should be aware of that it includes no protection mechanisms, which means that if as arguments not list of forward, but a list of source rules, or an inconsistent list of matches is passed, the method doesn't throw an error message, but performs the evaluation, just with a result value "false". This is shortcoming is assumed to be fixed in future work(V. 1).

The working mechanism of *TGGapplyFWRuls*() is illustrated in Image 37.

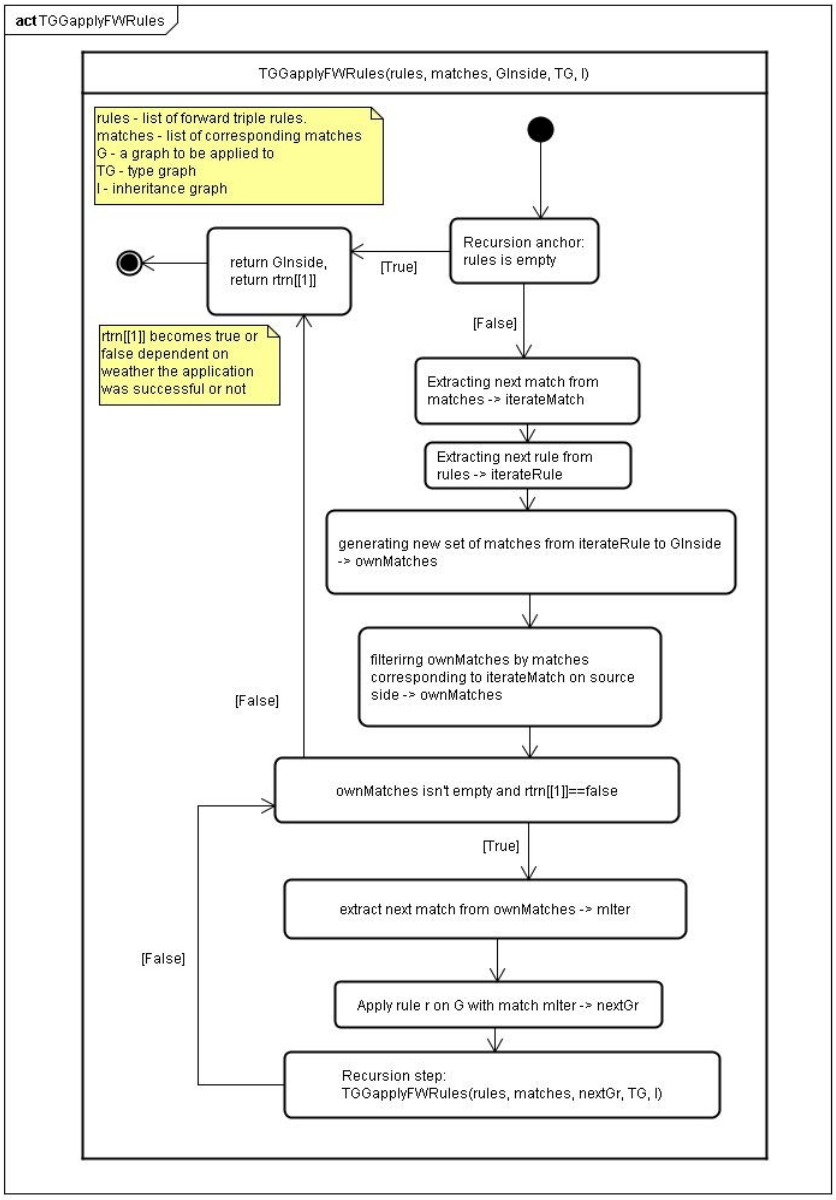


Image 37: working mechanism of TGGapplyFWRules()

First the return value and return graph are initialised, with "false" and the input graph respectively. The recursion anchor is reached when the passed rules list is empty. If its not the case next triple rule and corresponding source match are extracted from input lists(Code 34).

```
If[rulesInside != {},
 iterateRule = Last[rulesInside];
 iterateMatch = Last[matchesInside];
 rulesInside = Delete[rulesInside, -1];
 matchesInside = Delete[matchesInside, -1];
```

Code 34

Then a new set of matches from current forward rule to current triple graph from the input is generated by the use of *TGGMatches()*(3.9.). The resulting matches are compared in the source part with the input match to ensure the procedure to be source consistent (Definition 5). The comparison is performed by the use of select function and "Wild-card"(III. 1.1.) on the whole list of matching avoiding iteration through the list(Code 35).

```
ownMatches = TGGmatches[iterateRule, GInside, TG, I];
ownMatches = Select[ownMatches, #[[1]] == iterateMatch[[1]] &];
```

Code 35

However, not all iterations can be avoided. Next by the use of a "while" loop current input rule is applied to the input graph using first match from the result list in Code 35. The graph gained from this procedure is passed to *TGGapplyFWRules()*, together with the rest of the input rules list and input matches list – performing in this way the recursion step. In case that used match leads to a triple graph on which remaining rules in the input sequence couldn't be applied (which is the case if a value "false" is returned by *TGGapplyFWRules()* as a second argument), the "while" loop ensures that a backtracking will be performed and the input rule will be applied to the input graph by the use of an other match from in Code 35 generated list, since the loop is terminating only in case the match list is empty or all further recursion steps where performed successfully(Code 36).

```
While[ownMatches != {} && rtrn[[1]] == False,
 (*apply next rule with current match*)
 mInter= First[ownMatches];
 nextGr = TGGapplyRuleIndexed[iterateRule,mInter, GInside];
 (*recursion stepp to next rule*)
 rtrn = TGGapplyFWRules[rulesInside, matchesInside, nextGr, TG, I];
 ownMatches = Delete[ownMatches, 1]
 ],
```

Code 36

In such a way *TGGapplyFWRules()* applies forward rules in a predefined order to a source graph, with notion of source consistency till no rules left to apply, i.e. corresponding connection and target graphs were generated – the graph transformation performed.

TGGapplyFWRulesNM() performs similar actions like *TGGapplyFWRules(),* yet without comparing the generated match sequence with the comatches from source sequence, i.e. without checking source consistency and with an other arguments input scheme (first a list with indexes is given, specifying the order of forward rules to be applied, secondly all forward rules are passed, then the graph to be applied on, the type graph and inheritance graph).

3.8. Model Integration Realisation

Model integration similarly to model transformation part of the implementation consists of two methods, which convert a dpo rule into a source-/target- rule or an integration rule (*TGGSTRule()* and *TGGIntegrationRule()* respectively); two corresponding methods, which apply *TGGSTRule()* and *TGGIntegrationRule()* to lists of rules (*TGGSTRules()* and *TGGIntegrationRules*); a method to convert a triple graph to a source-/target- triple graph (*TGGSTGraph()*) and a main method *TGGapplyInegrationRules()* for applying a sequence of integration rules to a source-/target- triple graph.

TGGSTRule

Takes dpo rule (3.6.1) and a list of application conditions as argument, converts the dpo rule into a source-/target- rule and returns the result in form of a dpo rule(3.6.1).

The construction of the *TGGSTRule*() resembles very much the construction of *TGGSourceRule()*(3.7.): from the input dpo rule source and target components of left- and right-hand sides of the rule are extracted, out of the extracted components and some empty graph(for connection parts) new triple graphs are generated and passed to *TGGmakeRuleInclusion(),* so that the output of the method is a valid dpo rule (3.6.1) with empty connection graphs(Code 37).

```
{LS, sl, LC, tl, LT} = L;
{RS, sr, RC, tr, RT} = R;
sl = {{}, {}}; sr = {{}, {}};
tl = {{}, {}}; tr = {{}, {}};
LC = makeTypedGraph[{}, {}, TGC];
RC = makeTypedGraph[{}, {}, TGC];
LG = TGGmakeGraph[LS, sl, LC, tl, LT];
RG = TGGmakeGraph[RS, sr, RC, tr, RT];
rule = TGGmakeRuleInclusion[LG, RG, completeACs]];
```

Code 37: TGGSTRule

TGGIntegrationRule

Takes dpo rule (3.6.1) and a list of application conditions as argument, converts the dpo rule into an integration rule and returns the result in form of a dpo rule(3.6.1).

On the code of *TGGIntegrationRule()* it can be demonstrated how near to the theoretical concepts the implementation is. According to Definition 7 to construct an integration rule from a triple rule, following steps have to be performed:

1) Source and target graphs from the left-hand side of the rule have to be exchanged with source and target rule from the left-hand side.
2) morphisms between source and target components of left- and right-hand sides of the rule have to be exchanged with identity morphisms.
3) Morphisms from connection graph to source and target graphs in the left-hand side of the rule have to be exchanged with concatenations of morphisms $s \circ s_L$ and $t \circ t_L$ according to Image 38.

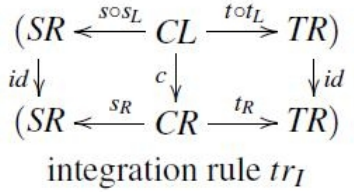


Image 38: integration Rule Construction

```
{LS, sl, LC, tl, LT} = L;
{RS, sr, RC, tr, RT} = R;
sl = compG[r[[1]], sl];
tl = compG[r[[3]], tl];
LG = {RS, sl, LC, tl, RT};
rule = TGGmakeRuleInclusion[LG, R, completeACs];
```

Code 38: integration rule implementation

The implementation of *TGGIntegrationRule()* is performed as follows(Code 38):

1) Source and target graphs from the left-hand side of the rule have are exchanged with source and target rule from the left-hand side (green marked in Code 38).
2) Since source and target graphs are equal the intersection of left- and right-hand source and target graphs will guarantee the generated morphisms from the interface triple graph to the right hand triple graph of the dpo rule (II. 1.5.) to be identity morphisms.
3) Morphisms from connection graph to source and target graphs in the left-hand side of the rule are exchanged with concatenations of morphisms $s \circ s_L$ and $t \circ t_L$ according to Image 38 (blue marked in Code 38).

The only difference is that *TGGmakeRuleInclusion()* returns a dpo rule, yet in the theory simple triple rules are used. But since integration rules are non-deleting rules the interphace triple graph of with *TGGIntegrationRule()* generated the rule is equal to the left-hand triple graph, which means that while application of this rule to any triple graph first part of Definition 9 will be skipped, which makes the output of *TGGIntegrationRule()* to a simple triple rule corresponding the theory.

TGGapplyInegrationRules

becomes a set of integration rules(output of TGGIntegrationRules()), a set of source-/target- matches, a source-/target- triple graph(output of TGGSTGraph()), a type graph and an inheritance graph as input. The method applies the integration rules from the input sequence to the input source-/target- graph. The result returned by the method is a triple graph and a boolean value, which states weather the application procedure was performed successfully.

Same as *TGGapplyFWRules()*(3.7.) is *TGGapplyIntegrationRules*() a recursive method and that it has also a similar to *TGGapplyFWRules()* working mechanism can be seen from diagram in Image 39. The difference is only that after the matches from in current recursion step focused rule and graph are generated, they are filtered by comparing not only to source but also to target parts of corresponding matches from the input sequence.

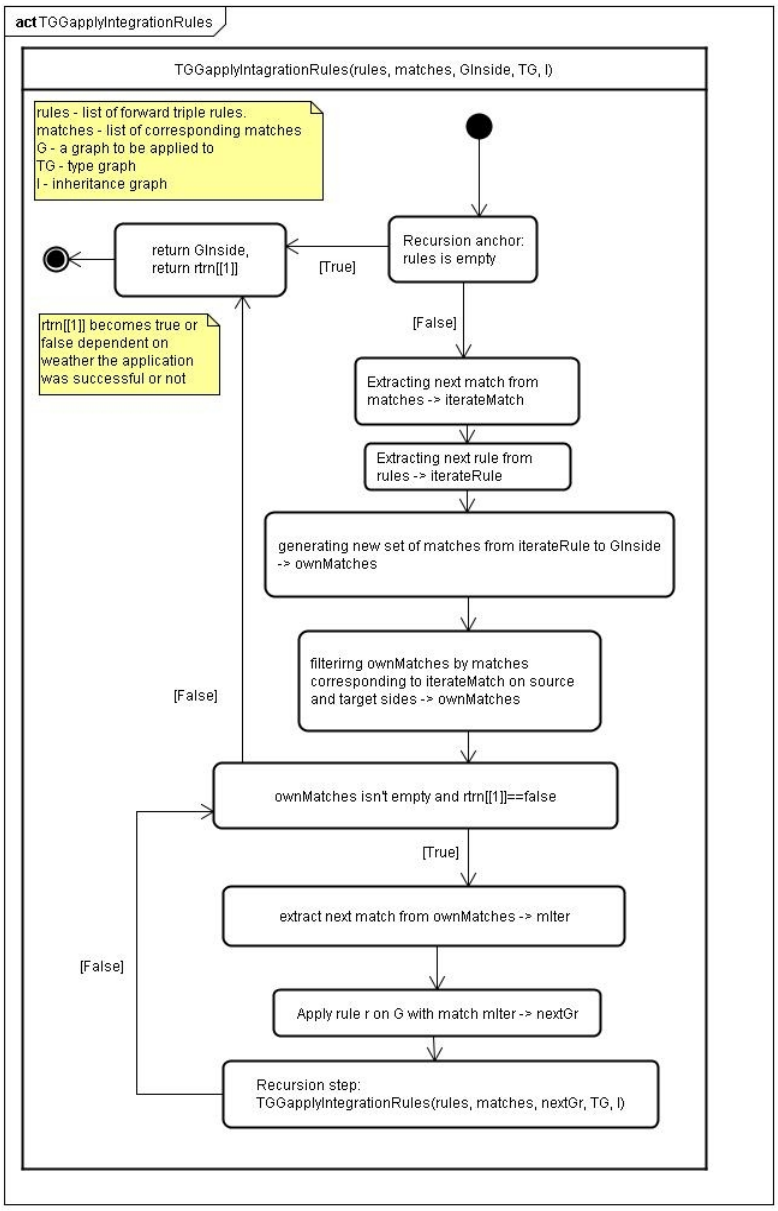


Image 39: working mechanism of TGGapplyIntagrationRules()

3.9. Match search functionality for Triple Rule on Triple Graph

Matching is one of major aspects in triple graph application concept. As mentioned in Section II. 1.2. Triple Rule and 3.5 Applying a Triple Rule on a Triple Graph a match is needed to define nodes and edges in graphs, which will get changed by the rule.

Normally matches are defined by the user himself. Only the user possess detailed information about the model (graph) on which current rule is being applied. Yet, in process of triple graph transformation realisation the necessity of match search automation occurred. There are two main spheres, where this approach is needed: source and source-/target- sequence search, transformation sequence and integration sequence application.

1. Source and source-/target- sequence search (3.10.) builds up a huge search tree during its execution. The search branches into various application variants. The amount and kinds of this variants depend on the possible application matches generated for the state in which the source (source-/target-) search is. The amount of matches grows exponential with the amount of nodes in graphs and amount of rules in the search sequence. On one hand is the source (source-/target-) search an automated method, which is designed to work without user intervention. On the other hand is in some cases, because of high branching degree, a manual match estimation near to impossible.
2. In forward (integration) sequence application source (source-/target-) sequences define the source (source-/target-) components of the match, yet not the whole match. There are two concepts to realize a method, which would search for a proper match in this case. One is to take available match components and complement the missing ones. Second is to generate matches in a normal way and filter for known components afterwords. In case of forward (integration) sequence application the second approach is used. The reason for it is the possibility to use the same method for both tasks: the search and the application.

TGGmatches

becomes a dpo rule(output of TGGmakeRuleInclusion()), triple graph(output of TGGMakeGraph), type graph and inheritance graph as argument. Calculates matches from input triple rule to input triple graph and checks the matches to be consistent. Returns a list of generated triple matches. Every triple match is a list of three elements, corresponding to source, connection and target matches. Every match is a list of two elements: nodes to match and edges to match. Every list of nodes or edges is an indexed list(III. 2.2. "Morphism realisation").

First *TGGmatches()* performs a splitting of the input data into source-, connection and target parts (Code 39).

```
{LS, sL, LC, tL, LT} = L;
{KS, sK, KC, tK, KT} = K;
{RS, sR, RC, tR, RT} = R;
{TGS, sTG, TGC, tTG, TGT} = TG;
{IS, sI, IC, tI, IT} = I;
{GS, sG, GC, tG, GT} = G;
{lS, lC, lT} = l;
{rS, rC, rT} = r;
```

Code 39

Split data is summarised according to the affiliation to source-, connection or target components and passed to *calculateMatches()*. *calculateMatches()* passes the data to *matchesSimple()* calculates possible matches from a production to a graph. In that way sets of matches for every component(source, connection, target) are generated. Additionaly, because *matchesSimple()* returns many redundant entries, the method *DeleteDuplicates()(III. 1.2.)* is applied to the result sets of *calculateMatches()*(Code 40).

```
matchesS = DeleteDuplicates[
 calculateMatches[{LS, lS, KS, rS, RS}, GS, {TGS[[1]], IS[[1]]}]];
matchesC = DeleteDuplicates[
 calculateMatches[{LC, lC, KC, rC, RC}, GC, {TGC[[1]], IC[[1]]}]];
matchesT = DeleteDuplicates[
 calculateMatches[{LT, lT, KT, rT, RT}, GT, {TGT[[1]], IT[[1]]}]];
```

Code 40

In next step the method *isGluingCondition()(III. 2.2.)* is applied to the matches, which were generated in previous step. Since there are sets of matches to check *isGluingCondition()* is applied in combination with the method *Map()*(III. 1.2.) and because of interest are only the matches fulfilling the gluing condition the method *If()* is used(Code 41).

```
matchesS = Map[
 If[isGluingCondition[GS[[1]], #, lS, KS[[1]], LS[[1]]] == True, #, {}] &,
  matchesS];
matchesC =  Map[
 If[isGluingCondition[GC[[1]], #, lC, KC[[1]], LC[[1]]] == True, #, {}] &,
 matchesC];
matchesT =  Map[
 If[isGluingCondition[GT[[1]], #, lT, KT[[1]], LT[[1]]] == True, #, {}] &,
 matchesT];
```

Code 41: matching: gluing condition check

```
result = Tuples[{matchesS, matchesC, matchesT}];
```

Code 42

Matches, which are returned by *TGGmatches()* are expected to be in form of a three element list. This list has to consist of one source, one connection and one target match. That for matches, which remained after the verification in Code 41 are mixed into three element lists by the use of the method *Tuples()*(III. 1.2.).

The gluing condition check performed in Code 41 verifies the match being consistent only in the context of a transformation ([EEP+06] "Definition 5.2 (transformation)") performed in the world of simple graphs. To verify the consistency of matches in the triple graph environment, the matches for source, connection and target components have to be inspected not only individually, but also in combination with each other. Such a consistency check is performed in *propperMatch()*. *PropperMatch()* in combination with *continueProperMatchINC() and propperMatchFromOpposite()* checks weather a match calculated for the source part of performed rule application is corresponding to some matched in the connection part or weather for a target match a connection match exists. *PropperMatch()* returns a boolean value. This value is the criteria for the *Select()*(III. 2.2.) method in the end of *TGGmatches()* to include or to exclude a match from the generated set from the result set. Which can be seen in Code 43.

```
result = Select[result, propperMatch[{LS, sL, LC, tL, LT}, #, G] == True &];
```

Code 43:

3.10. Automated source and source-/forward- sequence search

Triple graph transformation and integration theory describes a certainly interesting approach. But for this approach to be beneficial in practical use, there is one more task to accomplish. In sections 3.7. and 3.8. the implementation of triple graph transformation and integration were introduced. Both implemented methods get a sequences of triple rules (source or source-/target-). The whole triple graph transformation / integration theory bases on the existence of some sequence of triple rules, which were used to build up the given model. This means, that to perform a real transformation or integration the application has to parse the input model for a source sequence or a source-/target- sequence.

ApplyFun() was implemented with this purpose. This method uses deletion rules in the parsing process. Deletion rules are generated by applying *TGGdelrule()* to a dpo rule.

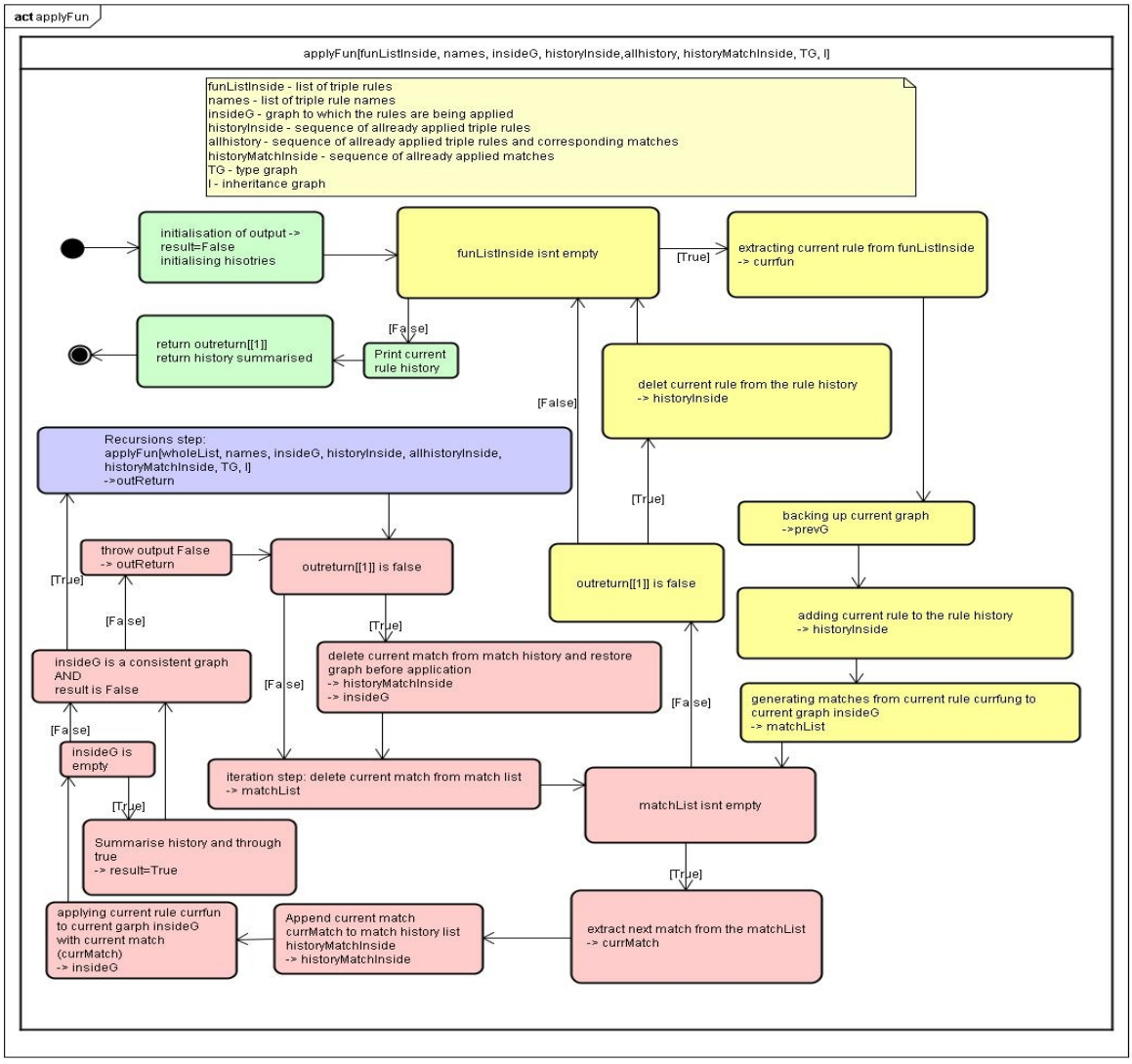


Image 40: working mechanism of applyFun()

TGGdelrule

takes a dpo triple rule(output of TGGmakeRuleInclusion()(3.6.2)) as an argument and replaces the left hand side graph with the right hand one, and the corresponding morphisms. The output is a dpo triple rule, which deletes the changes made to a graph by given input dpo rule, if applied with its comatches.

TGGmakeDelRules() applies *TGGdelrule()* on a list of dpo rules and returns a list of corresponding deleting rules.

ApplyFun

takes as arguments a list of deleting dpo triple rules (output of TGGmakeDelRules()), a list of corresponding rule names, a triple graph to parse(output of TGGmakeGraph()(3.3.)), two empty lists (for recursion realisation), a type graph and an inheritance graph. The output is a list, consisting of a boolean value, a list of indexes, a list of matches and a summary list. The list of indexes is the order in which the rules from input list have to be applied to an empty graph with the matches from the list of matches to generate the input graph.

ApplyFun() parses the input graph by applying the deletion rules from the input list to it. Before performing any operations the input graph is backed up. The method iterates over the input list of deletion rules. For each rule and the input graph a set of matches is generated. After all marches are inspected for focused rule, next rule in the list is chosen (yellow actions in Image 40 and Code 44).

```
(*iterating through all rules*)
While[funListInside ≠ {},
  (*itaration step→extracting next rule*)currfun = First[funListInside];
  funListInside = Delete[funListInside, 1];
  (*backing up current graph*)prevG = insideG;
  (*extracting the position of current applied rule and adding it to the rule history*)
  historyInside = Flatten[Append[historyInside, Flatten[Position[wholeList, currfun]]]];
  (*generating matches for current function and current graph*)
  matchList = TGGmatches[currfun, insideG, TG, I];
  (*iterating over the matches*)
  While[matchList ≠ {},
   ...
  ];
  (*if output is false*)
  If[outReturn[[1]] == False,
   (*delete rule from history*)
   historyInside = Delete[historyInside, -1]
  ];
 ];
```

Code 44: applyFun: input deletion dpo rules list loop

Each deletion rules is applied with each generated match. This is realized by iteration over the matches (red actions in Image 40). In the matches loop iteration step a match is chosen and written into a history variable. Then the rule is applied (Code 45).

```
(*extracting next match*)
currMatch = First[matchList];
(*saving the match in the history*)
historyMatchInside = Append[historyMatchInside, currMatch];
(*applying current rule with current match*)
insideG = TGGapplyRuleIndexed[currfun, currMatch, insideG];
```

Code 45

After rule application it is checked weather the empty triple graph has been reached. If it is the case the output variable "outReturn" is filled with a list. The list contains "true" as first element, a sequence of applied rules and a sequence of applied matches(Code 46).

```
If[V[insideG[[1]][[1]]] == 0 && V[insideG[[3]][[1]]] == 0 && V[insideG[[5]][[1]]] == 0,
 (*append match and rule history to the output history*)
 allhistoryInside = Append[allhistoryInside, {historyInside, historyMatchInside}];
 (*initialise the output with the match and rule history and signalise sequence to be found*)
 outReturn = {True, historyInside, historyMatchInside, allhistoryInside}, {}];
```

Code 46

Further, the graph generated in Code 45 is checked for being consistent triple graph, by a special method *checkTGG()*. It is an inconsistency if, for example, source and connection graphs contain no nodes, but the morphism between them isn't empty. If the graph is consistent, but not empty the recursion step is performed and *ApplyFun()* is applied with same dpo deletion rules list, yet with the new graph (blue action in Image 40 and Code 47).

```
If[checkTGG[insideG, prevG] == True && outReturn[[1]] ≠ True,
 (*go into recursion step with next graph*)
 outReturn = applyFun[wholeList, names, insideG, historyInside, allhistoryInside, historyMatchInside, TG, I],
 (*else→return history and false*)
 outReturn = {False, historyInside, historyMatchInside, allhistoryInside}];
```

Code 47: applyFun: recursion step

If after performed manipulations the value of the first element in "outReturn" list still stays "False", the match entry is erased from history list and the in Code 44 backed up graph is restored. The deletion of used match from the iteration list ends the match loop (Code 48).

```
If[outReturn[[1]] == False,
 (*delete current match from history*)
 historyMatchInside = Delete[historyMatchInside, -1];
 (*restore previous graph*)
 insideG = prevG;];
(*iteration step→deleting current match from match list*)
matchList = Delete[matchList, 1];];
```

Code 48

Thanks to the use of DPO rules, *applyFun()* is not fixed to the search only for the source or only for the target sequence. An other advantage is that *applyFun()* performs the pattern search not by generating whole set of possible graphs from given rules, but aims directly the build up sequence. The algorithms achieves this by searching for an empty graph as recursion anchor and applying deletion DPO rules in recursion step.

IV. CASE STUDIES

1 Company employee interdependency example

In the first case study the example class- to data base conversion diagram from II. 1.3. is taken up. The rules from II. 1.3. are defined, so that they can be used by the application. The diagram is first manually build up to prove the possibility of gaining it by the use the triple rules. Then an automated transformation and integration are performed. The resulting graphs of the transformation and integration are compared to the manually constructed triple graph to ensure that these operations return a correct result.

To evaluate the entity relationship case study the notebook file "TripleGraphTest_classentity_withNeeds.nb" in root Folder was created.

First step of evaluation in "TripleGraphTest_classentity_withNeeds.nb" is the loading of additional packages (Code 49).

```
Needs["TripleGraph`"]
Needs["Matching`"]
Needs["ClassEntityTG`"]
Needs["ClassEntityTGGRules`"]
Needs["ClassEntityTripleGraphsForRules`"]
```

Code 49

Following packages are being used in this case sturdy:

- TripleGraph.m

  Contains the definition of methods, which implement the triple graph transformation theory, i.e. *TGGMakeGraph(), TGGmakeGraphL(), TGGmakeRule(), TGGmakeRuleInclusion(),* etc. A detailed description of this methods can be found in sections III. 3.3. - III. 3.6. and III. 3.10. .

- Matching.m

  Contains the definitions of methods: *TGGmatches(), propperMatch(), calculateMatches()*, etc. This is a package from the implementation of III. 2 Realisation of attributed Graph-transformation in Mathematica(by Jochen Adamek), which has been extended by methods needed in the triple graph transformation theory. Methods from Matching.m are used for automatic match calculations between triple rules and triple graphs. These methods are described in detail in section III. 3.9. .

- ClassEntityTG.m

Contains the definitions of type graphs for source, connection and target parts of the models, which are used in the case study. This means that node and edge types are defined and the inheritance graphs created. In Code 50 can be observed how this procedure is performed for the class diagram model. *Class$TGNodes* is a list, which contains an entry for every node type from class diagram language, used in the case study. To *Class$TG* a graph is assigned consisting of *Class$TGNodes* elements as nodes and all possible edges between them. *Class$I* represents the inheritance graph of the model. In case of class diagram used for this case study the inheritance graph is flat (has only one dimension). *Class$TGI* summarises the type graph and inheritance graph in one list. Similar are the definitions for connection and entity-relationship diagrams.

```
Class$TGNodes := {"ClassClass", "ClassAssociation", "ClassAttribute",
  "ClassPrimitiveDataType"}
Class$TG := makeGraph[Class$TGNodes,
  {{"Class$attrs", {"ClassClass", "ClassAttribute"}},
   {"Class$type", {"ClassAttribute", "ClassClass"}},
   {"Class$src", {"ClassAssociation", "ClassClass"}},
   {"Class$dest", {"ClassAssociation", "ClassClass"}},
   {"Class$type", {"ClassAttribute", "ClassPrimitiveDataType"}},
   {"Class$parent", {"ClassClass", "ClassClass"}}}]
Class$I := makeGraph[Class$TGNodes, {}]
Class$TGI := {Class$TG, Class$I}
```

Code 50:class diagram: type graph generation

- ClassEntityTGGRules.m

  Contains the definitions for graphs and triple graphs, which are needed to build up the triple rules used in this case study. For instance, to build up the rule subClass2Table (II. 1.2. Image 4) six graphs or two triple graphs are needed:

  1) Left triple graph (purple in Code 51) consists of:
     (a) source graph (class diagram part): node of type “ClassClass” (blue in Code 51)
     (b) connection graph: node of type “ClassTableRel” (green in Code 51)
     (c) target graph (entity-relationship diagram part): node of type “EntityTable” (red in Code 51).

```
class2TableClass := makeTypedGraph[{{"c1", "ClassClass"}}, {}, Class$TG];
class2TableConnection := makeTypedGraph[{{"ctr1", "ClassTableRel"}},
   {}, Connection$TG];
class2TableEntity := makeTypedGraph[{{"t1", "EntityTable"}}, {}, Entity$TG];
class2Table := TGGmakeGraph[class2TableClass, {{{"ctr1", "c1"}}, {}},
   class2TableConnection, {{{"ctr1", "t1"}}, {}}, class2TableEntity];
```

Code 51

2) Right triple graph (purple in Code 52) consists of:

   (a) source graph: two "ClassClass" nodes, connected to each other with an edge of type "Class$parent" (blue in Code 52)

   (b) connection graph: two nodes of type "ClassTableRel" (green in Code 52)

```
subclass2TableClass := makeTypedGraph[{{"c1", "ClassClass"}, {"c2", "ClassClass"}},
   {{"p1", {"c2", "c1"}, "Class$parent"}}, Class$TG];
subclass2TableConnection :=
  makeTypedGraph[{{"ctr1", "ClassTableRel"}, {"ctr2", "ClassTableRel"}},
   {}, Connection$TG];
subclass2TableEntity := makeTypedGraph[{{"t1", "EntityTable"}}, {}, Entity$TG];
subclass2Table := TGGmakeGraph[subclass2TableClass,
   {{{"ctr1", "c1"}, {"ctr2", "c2"}}, {}}, subclass2TableConnection,
   {{{"ctr1", "t1"}, {"ctr2", "t1"}}, {}}, subclass2TableEntity];
```

Code 52

   (c) target graph: a node of type "EntityTable" (red in Code 52).

Similar definitions are made for graphs and triple graphs used to build up the rules *Class2Table, PrimaryAttribute2Column* and *Association2ForegnKey.*

- ClassEntityTripleGraphsForRules.m

  includes the actual definitions of the rules *Class2Table, SubClass2Table, PrimaryAttribute2Column* and *Association2ForegnKey (Code 53).*

```
class2TableRule = TGGmakeRuleInclusion[empty, class2Table, {}];
subclass2TableRule = TGGmakeRuleInclusion[class2Table, subclass2Table, {}];
primaryAttribute2ColumnRule =
  TGGmakeRuleInclusion[primaryAttribute2ColumnL, primaryAttribute2ColumnR, {}];
Association2ForeignKey = TGGmakeRuleInclusion[Association2ForeignKeyL,
   Association2ForeignKeyR, {}];
Association2ForeignKeyRule =
  TGGmakeRuleInclusion[Association2ForeignKeyL, Association2ForeignKeyR, {}];
```

Code 53

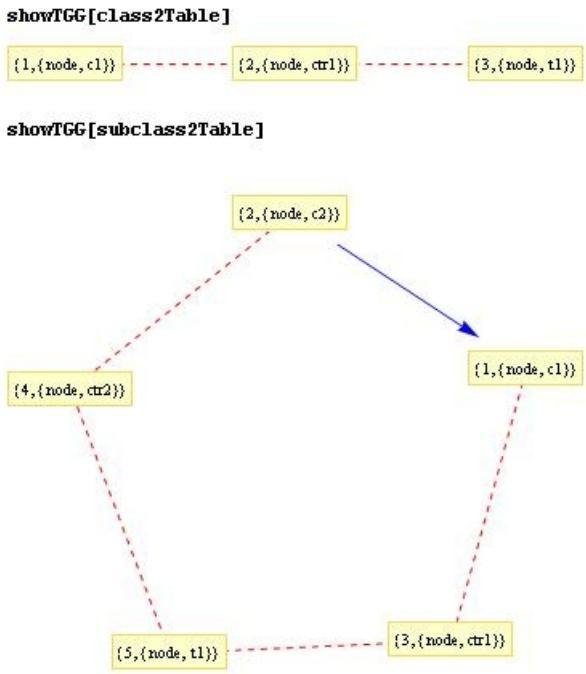
ClassEntityTripleGraphsForRules.m additionally contains a list *allRulesClassEntity*, which includes all of the defined triple rules, and a list *allRulesClassEntityNames* of corresponding rule names(Code 54).

```
allRulesClassEntity := {class2TableRule, subclass2TableRule,
   primaryAttribute2ColumnRule, Association2ForeignKeyRule};

allRulesClassEntityNames := {"class2TableRule", "subclass2TableRule",
   "primaryAttribute2ColumnRule", "Association2ForeignKey"};
```

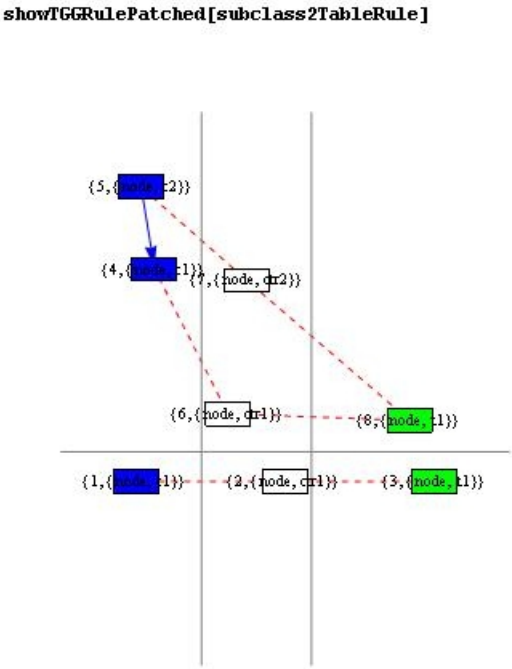
Code 54

After loading additional packages in "TripleGraphTest_classentity_withNeeds.nb" the loaded components are visualised.

The Visualisation is performed by the use of methods *showTGG()* and *showTGGRulePatched()*. This methods are extensions of Mathematica build in method *GraphPlot()*(III. 1.2.). *showTGG(), showTGGRulePatched()* as well as visualisation methods *showTGGPatched(), showTGGRuleAll(), showTGGRulePureAll()* and *showTGGRulePurePatched()* were developed during implementation of the practical part of this theses too, yet couldn't be brought into the description, because of lack of time. The general functionalities of *showTGG()* and *showTGGRulePatched()* are following:



Code 55: Visualisation: class3Table and subclass2Table



Code 56: Visualisation: subclass2TableRule

showTGG

visualises a triple graph. The source graph edges are coloured blue in the visualisation, the target graph edges are coloured green, but the morphisms are visualised as red dashed arrows.

showTGGRulePatched

visualises a triple rule.

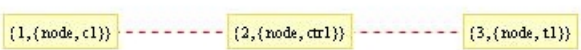
- *If the left-hand triple graph is empty – the visualisation consists of six cells: three upper ones realize the left hand triple graph, three lower ones realize the right hand triple graph.*
- *If the left-hand triple graph isn't empty – the visualisation consists of six cells: three lower ones realize the left hand triple graph, three upper ones realize the right hand triple graph.*

In both cases blue nodes belong to source graphs, white to connection graphs and green to target graphs.

All triple graphs from "ClassEntityTGGRules.m" as well as triple rules from "ClassEntityTripleGraphsForRules.m" are visualised in "TripleGraphTest_classentity_withNeeds.nb". For example, the rule *subclass2tableRule(),* which was constructed in Code 51 and Code 52 is also visualised in "TripleGraphTest_classentity_withNeeds.nb". The visualisation includes the left- and right- hand triple graphs (*class2table* and *subclass2table*) (Code 55) and the triple rule *subclass2tableRule* itself (Code 56).

Triple graphs and triple rules visualisations are followed by a manual constructed triple rule execution sequence from *empty* (which is the empty triple graph) to *G5*. This is done in order to prove that under explicitly defined conditions triple graph *G5* is constructed by the use of triple rules from the list *allRulesClassEntity. G1* is the result of *class2TableRule()* applied to *empty* with an empty match(Code 57), *G2* - *class2TableRule()* applied to *G1*, etc.

```
G1 = TGGapplyRule[class2TableRule, {{{}, {}}, {{}, {}}, {{}, {}}}, empty];

showTGG[G1]
```

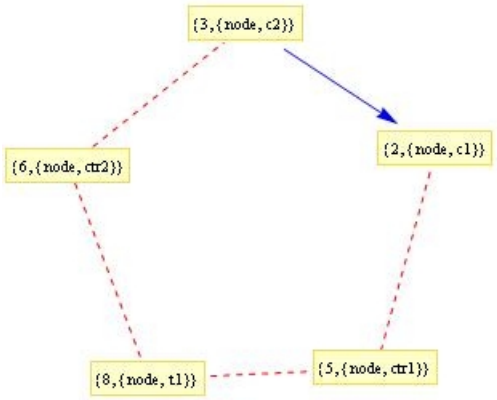


Code 57

```
mG2 = TGGmatches[subclass2TableRule, G2, SCT$TG, SCT$I]

{{{{1}, {}}, {{1}, {}}, {{1}, {}}}, {{{2}, {}}, {{2}, {}}, {{2}, {}}}}
```

Code 58

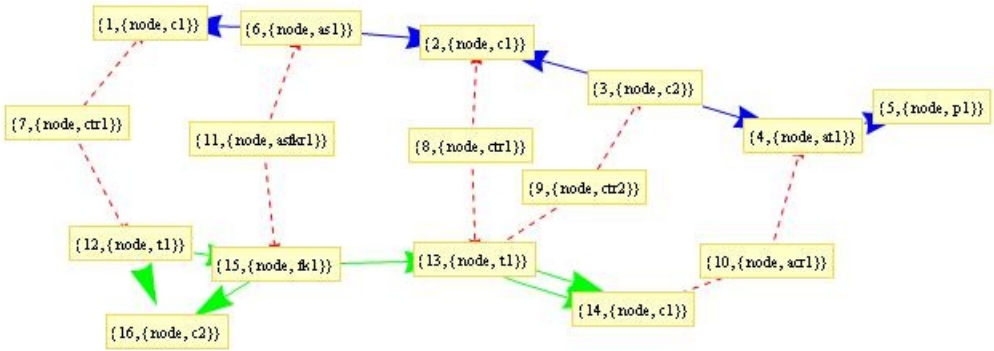
```
G3 = TGGapplyRuleIndexed[subclass2TableRule, mG2[[2]], G2];

showTGG[G3]
```



Code 59

Beginning with *subclass2TableRule()* application, the matches are calculated by the use of the method *TGGmatches()*(Code 58) and applied by the use of the method *TGGapplyRuleIndexed()*. Yet the choice of the matches to use stays determined manually (red underlined in Code 59).

```
showTGG[G5]
```



Code 60: result graph G5 after manual triple rule application

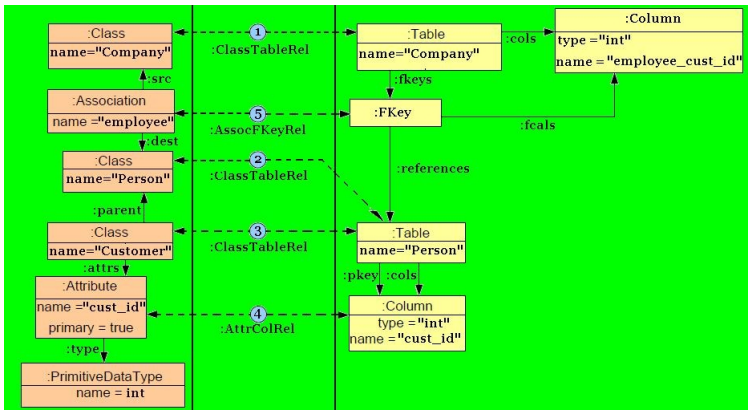


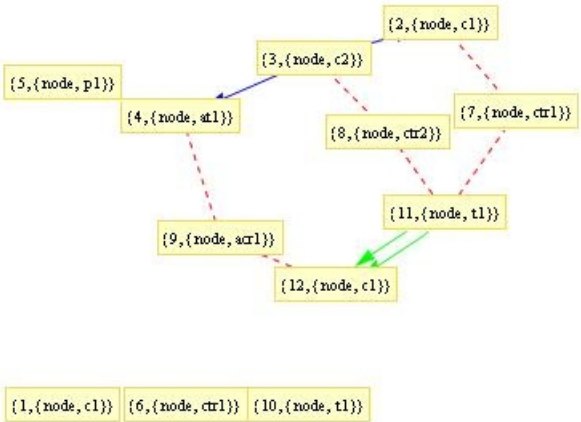
Image 41: final triple graph in transformation/integration example from II. 1.3. /II. 1.4.

The resulting graph *G5*(Code 60) corresponds to G_{10} from II. 1.4. Table 4 (Image 41).

The sequence used for constructing *G5* is $empty \overset{class2TableRule}{\Rightarrow} G1 \overset{class2TableRule}{\Rightarrow} G2 \overset{subclass2TableRule}{\Rightarrow} G3 \overset{primaryAttribute2ColumnRule}{\Rightarrow} G4 \overset{Association2ForeignKeyRule}{\Rightarrow} G5$.

Further in the deletion rules are tested. It is done by transforming every rule in a corresponding to it deletion rule and applying them in inverse sequence $G5 \overset{Association2ForeignKeyRuleDel}{\Rightarrow} G4 \overset{primaryAttribute2ColumnRuleDel}{\Rightarrow} G3 \overset{subclass2TableRuleDel}{\Rightarrow} G2 \overset{class2TableRuleDel}{\Rightarrow} G1 \overset{class2TableRuleDel}{\Rightarrow} empty$ to *G5*. In Code 61 it is shown how this procedure is performed to the rule *Association2ForegnKey*. The matches, which are used for application of *Association2-ForegnKeyDel* are also here generated in automatic way by the use of *TGGmatches()*(III. 3.9.)(green underlined in Code 61). Yet also in deletion rules case generated matches are applied manually (red underlined in Code 61).

```
(* Backward manual TGT-sequence *)

Association2ForeignKeyRuleDel = TGGdelrule[Association2ForeignKeyRule];

mG51 = TGGmatches[Association2ForeignKeyRuleDel, G5, SCT$TG, SCT$I];

G41 = TGGapplyRuleIndexed[Association2ForeignKeyRuleDel, mG51[[1]], G5];

showTGG[G41]
```



Code 61: Association2ForeighKeyRuleDel: deletion rule generation and application

As expected applying deletion rules in above mentioned sequence ends up in an empty graph *empty.* This result has proven, that it is possible to apply deletion rules to a graph in a reverse order to the triple rule build up sequence of this graph and get an empty graph.

1.1.1 Automated triple graph transformation

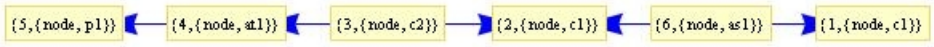
Further in "TripleGraphTest_classentity_withNeeds.nb" an automatic transformation is being performed. For this reason the package "TrippleGraphTransformation.m" is being loaded(Code 62). This package contains methods like *TGGSourceRules(), TGGForwardRules(), TGGapplyFWRules,* etc. These methods were developed for performing triple graph transformation. In Section III. 3.7. the implementation of triple graph transformation is described in detail.

```
Needs["TrippleGraphTransformation`"]
```

Code 62

The task of a triple graph transformation is to transform a source model into a target model. That for, first *G5* is transformed, by the use of *TGGSourceGraph(),* into a source graph *G5S* (Code 63).

```
G5S = TGGSourceGraph[G5, Class$TG, Entity$TG];

showTGG[G5S]
```



Code 63: Case study 1: source graph

G5S represents the class diagram, which is transformed during the experiment in "TripleGraphTest_classentity_withNeeds.nb" into a corresponding entity-relationship diagram.

```
sourceRules = TGGmakeDelRules[TGGSourceRules[allRulesClassEntity, Class$TG, Entity$TG]];

forwardRules = TGGForwardRules[allRulesClassEntity, Class$TG, Entity$TG];
```

Code 64:

To perform the transformation according to the triple graph transformation theory (II. 1.3.) some more components are needed. The corresponding to the source graph – connection and target graphs are build up by the execution of a forward rule sequence. And the forward rule sequence is determined by a source rule sequence. This means, that aside from the source graph *G5S* source and forward rules have to be generated(Code 64).

```
...
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
...
```

Code 65: applyFun: output during execution

The triple graph transformation theory doesn't provide any information about the generation technique of the source rule sequence. Therefore the in section III. 3.10. introduced method *applyFun()* can be used. a*pplyFun()* generates the source sequence by applying deletion source rules to the source model. That is the reason, why in *Code 64* generated source rules are transformed in corresponding deleting rules.

During the execution of *applyFun()* a lot of textual output is being produced(Code 65). This happens because *applyFun()* is a recursive method. Each line of output represents the result sequence, which is calculated at a certain recursion step. The user can see in that way the whole track, which led to the generation of the source sequence. Of direct importance is the last output string and the second element in the output list(green underlined in Code 66). These are the result parsing source sequence generated form *G5S* by the use of *applyFun()* and the deleting source rules.

```
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
{False, {4, 3, 2, 1, 1}, {{{{1, 2, 6}, {4, 5}}, {{}, {}}, {{}, {}}},
  {{{3, 4, 5}, {2, 3}}, {{}, {}}, {{}, {}}}, {{{2, 3}, {1}}, {{}, {}}, {{}, {}}},
  {{{2}, {}}, {{}, {}}, {{}, {}}}, {{{1}, {}}, {{}, {}}, {{}, {}}}},
 {{{4, 3, 2, 1, 1}, {{{{1, 2, 6}, {4, 5}}, {{}, {}}, {{}, {}}},
    {{{3, 4, 5}, {2, 3}}, {{}, {}}, {{}, {}}}, {{{2, 3}, {1}}, {{}, {}}, {{}, {}}},
    {{{2}, {}}, {{}, {}}, {{}, {}}}, {{{1}, {}}, {{}, {}}, {{}, {}}}}}}}
```

Code 66: applyFun(): final output

The fact, that the in Code 66 generated sequence correspond to manually established deletion rule sequence $G5 \overset{Association2ForeignKeyRuleDel}{\Rightarrow} G4 \overset{primaryAttribute2ColumnRuleDel}{\Rightarrow} G3 \overset{subclass2TableRuleDel}{\Rightarrow} G2 \overset{class2TableRuleDel}{\Rightarrow} G1 \overset{class2TableRuleDel}{\Rightarrow} empty$ proves, that automated source sequence search implemented in this thesis is successful, when applied in borders of this case study.

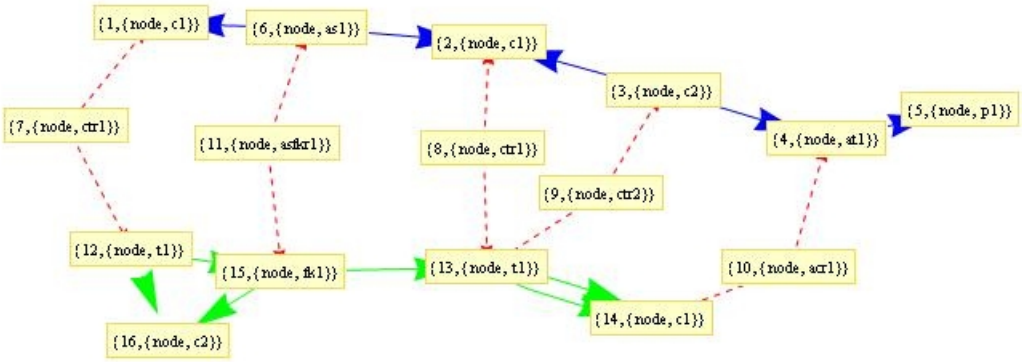
After establishing the source rule sequence finally the model transformation according to triple graph transformation theory (II. 1.3.) can be performed. Forward rules are applied to the source model *G5S* in by the triple rule source sequence defined order. This is performed by the use of *TGGapplyFWRules()(III. 3.7.)(Code 67).*

```
res = TGGapplyFWRules[Map[forwardRules[[#]] &, sourceSeq[[4]][[1]][[1]] ],
  sourceSeq[[4]][[1]][[2]], G5S, SCT$TG, SCT$I]
```

Code 67: performing triple graph transformation

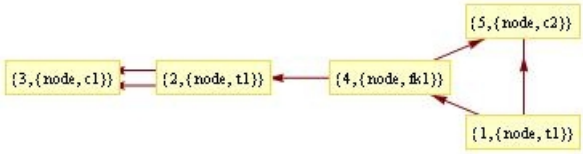
The automated transformation result triple graph (Code 68) represents the class diagram connected via a graph and two morphisms to the corresponding entity-relationship diagram. The fact that it corresponds with manually constructed graph *G5* from Code 60 proves, that automated triple graph transformation implemented in this thesis is successful, when applied in borders of this case study. To get the sought entity-relationship model the graph from Code 68 is reduced to the target graph(Code 69).

```
showTGG[res[[2]]]
```



Code 68: automated transformation result triple graph

```
show[res[[2]][[5]][[1]]]
```



Code 69: automated transformation result: entity-relationship model

1.1.2 Automated triple graph integration

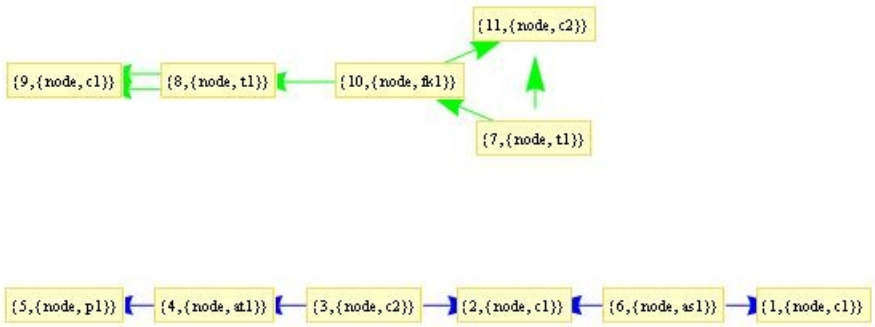
For performing an automated model integration according to the triple graph integration theory(II. 1.4.) first, like in triple graph transformation, additional methods are needed. Those methods are gained by loading "TrippleGraphIntegration.m"(Code 70) into the notebook "TripleGraphTest_classentity_withNeeds.nb". This package contains methods like *TGGSTRules(), TGGIntegrationRules(), TGGapplyInegrationRules,* etc. These methods were developed for performing triple graph integration. In Section III. 3.8. the implementation of triple graph integration is described in detail.

```
Needs["TrippleGraphIntegration`"]
```

Code 70

The task of a triple graph integration is to perform a connection between two models. For that reason, first the manually constructed test-model triple graph *G5* from Code 60 is transformed, by the use of *TGGSTGraph(),* into a source/target graph *G5ST* (Code 71).

```
G5ST = TGGSTGraph[G5, Class$TG, Entity$TG];

showTGG[G5ST]
```



Code 71: automated graph integration: source-/target- graph

G5ST represents the class and entity-relationship diagrams, which have to be connected by a third graph during the experiment automated triple graph integration in "TripleGraphTest_classentity_withNeeds.nb".

```
stDEL = TGGmakeDelRules[TGGSTRules[allRulesClassEntity, Class$TG, Entity$TG]];

inegrationRules = TGGIntegrationRules[allRulesClassEntity, Class$TG, Entity$TG];
```

Code 72

To perform the integration according to the triple graph integration theory (II. 1.4.) some more components are needed. The corresponding to the source-/target- graph – connection graph is build up by executing an integration rule sequence. The integration rule sequence is determined by a source-/target- rule sequence. This means, that aside from the source-/target- graph *G5ST* source-/target- and integration rules have to be generated(Code 72).

The triple graph integration theory doesn't provide any information about the generation technique of the source-/target- rule sequence. Therefore the in section III. 3.10. introduced method *applyFun()* can be used. a*pplyFun()* generates the source-/target- sequence by applying deletion source-/target- rules to the source-/target- model. That is the reason,

```
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
 {False, {4, 3, 2, 1, 1}, {{{{1, 2, 6}, {4, 5}}, {{}, {}}, {{1, 2, 4, 5, 3}, {1, 3, 4, 5, 6}}},
   {{{3, 4, 5}, {2, 3}}, {{}, {}}, {{2, 3}, {1, 2}}}, {{{2, 3}, {1}}, {{}, {}}, {{2}, {}}},
   {{{2}, {}}, {{}, {}}, {{2}, {}}}, {{{1}, {}}, {{}, {}}, {{1}, {}}}},
  {{{4, 3, 2, 1, 1}, {{{{1, 2, 6}, {4, 5}}, {{}, {}}, {{1, 2, 4, 5, 3}, {1, 3, 4, 5, 6}}},
     {{{3, 4, 5}, {2, 3}}, {{}, {}}, {{2, 3}, {1, 2}}}, {{{2, 3}, {1}}, {{}, {}}, {{2}, {}}},
     {{{2}, {}}, {{}, {}}, {{2}, {}}}, {{{1}, {}}, {{}, {}}, {{1}, {}}}}}}}
```

Code 73: applyFun() result: source-/target- sequence

why in *Code 72* generated source-/target- rules are transformed in corresponding deleting rules.

The result of *applyFun()* is a rule sequence (last output string and the second element in the output list(green underlined in Code 73)) which can be used to build up given source-/target- model. This is the result parsing source-/target- sequence, which is generated form *G5ST* by the use of *applyFun()* and the deleting source-/target- rules.

The fact, that the in Code 73 generated sequence correspond to manually established deletion rule sequence $G5 \overset{Association2ForeignKeyRuleDel}{\Rightarrow} G4 \overset{primaryAttribute2ColumnRuleDel}{\Rightarrow} G3 \overset{subclass2TableRuleDel}{\Rightarrow} G2 \overset{class2TableRuleDel}{\Rightarrow} G1 \overset{class2TableRuleDel}{\Rightarrow} empty$ proves, that automated source-/target- sequence search implemented in this thesis is successful, when applied in borders of this case study.

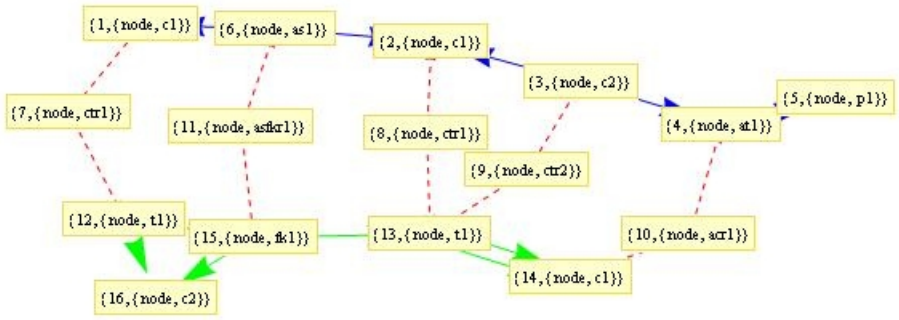
After establishing the source-/target- rule sequence finally the model integration according to triple graph integration theory (II. 1.4.) can be performed. Integration rules are applied to the source-/target- model *G5ST* in by the triple rule source-/target- sequence defined order. This is performed by the use of *TGGapplyIntegrationRules()(III. 3.8.)(Code 74).*

```
resST = TGGapplyInegrationRules[Map[inegrationRules[[#]] &, stSeq[[4]][[1]][[1]] ],
  stSeq[[4]][[1]][[2]], G5ST, SCT$TG, SCT$I]
```

Code 74: performing triple graph integration

The automated integration result triple graph (Code 75) represents the class diagram connected by a graph and two morphisms to the corresponding entity-relationship diagram. The fact that it corresponds with manually constructed graph *G5* from Code 60 proves, that automated triple graph integration implemented in this thesis is successful, when applied in borders of this case study.

```
showTGG[resST[[2]]]
```



Code 75: automated integration result triple graph

2 Car factory software example

In the second case study a model representing an car entry in a car factory software is used(Image 42). The simulation of triple garph transformation and integration is performed in the notebook “TripleGraphTest_classentity2_withNeeds.nb”.

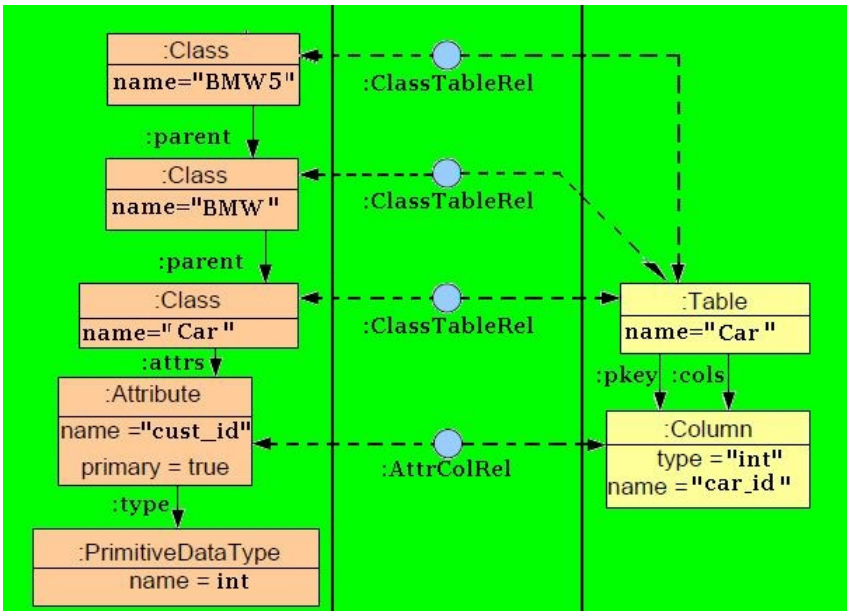
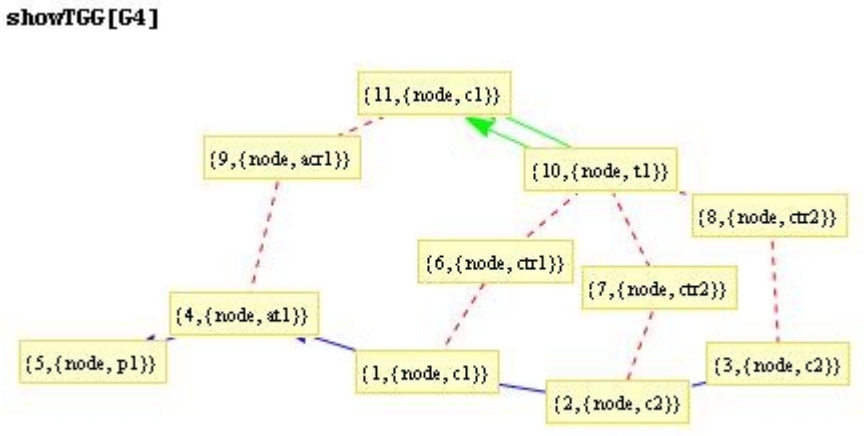


Image 42: Case study 2 models: Car factory

In the same way as in as in the first case study(IV. 2.1.1) to Image 42 corresponding triple graph is manually constructed by the use of in “ClassEntityTripleGraphsForRules.m” defined triple rules. The resulting triple graph is illustrated in Code 76.



Code 76: result graph G4 after manual triple rule application

2.1.1 Automated triple graph transformation

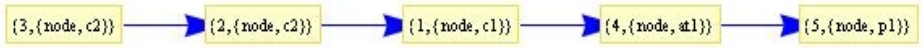
Further in "TripleGraphTest_classentity2_withNeeds.nb" an automatic transformation is being performed. For this reason the package "TrippleGraphTransformation.m" is being loaded(Code 77). This package contains methods like *TGGSourceRules(), TGGForward-Rules(), TGGapplyFWRules,* etc. These methods were developed for performing triple graph transformation. In Section III. 3.7. the implementation of triple graph transformation is described in detail.

```
Needs["TrippleGraphTransformation`"]
```

Code 77

The task of a triple graph transformation is to transform a source model into a target model. That for, first *G4* is transformed, by the use of *TGGSourceGraph(),* into a source graph *G4S* (Code 78).

```
G4S = TGGSourceGraph[G4, Class$TG, Entity$TG];

showTGG[G4S]
```



Code 78: Case study 1: source graph

G4S represents the class diagram, which is transformed during the experiment in "Triple-GraphTest_classentity2_withNeeds.nb" into a corresponding entity-relationship diagram.

```
sourceRules = TGGmakeDelRules[TGGSourceRules[allRulesClassEntity, Class$TG, Entity$TG]];

forwardRules = TGGForwardRules[allRulesClassEntity, Class$TG, Entity$TG];
```

Code 79:

To perform the transformation according to the triple graph transformation theory (II. 1.3.) some more components are needed. The corresponding to the source graph – connection and target graphs are build up by the execution of a forward rule sequence. And the forward rule sequence is determined by a source rule sequence. This means, that aside from the source graph *G4S* source and forward rules have to be generated(Code 79).

```
...
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
rules: {Association2ForeignKey, primaryAttribute2ColumnRule,
  subclass2TableRule, class2TableRule, class2TableRule}
...
```

Code 80: applyFun: output during execution

The triple graph transformation theory doesn't provide any information about the generation technique of the source rule sequence. Therefore the in section III. 3.10. introduced method *applyFun()* can be used. a*pplyFun()* generates the source sequence by applying deletion source rules to the source model. That is the reason, why in *Code 79* generated source rules are transformed in corresponding deleting rules.

During the execution of *applyFun()* a lot of textual output is being produced(Code 80). This happens because *applyFun()* is a recursive method. Each line of output represents the result sequence, which is calculated at a certain recursion step. The user can see in that way the whole track, which led to the generation of the source sequence. Of direct importance is the last output string and the second element in the output list(green underlined in

```
rules: {primaryAttribute2ColumnRule, subclass2TableRule, subclass2TableRule, class2TableRule}
 {False, {3, 2, 2, 1},
  {{{{1, 4, 5}, {3, 4}}, {{}, {}}, {{}, {}}}, {{{2, 3}, {2}}, {{}, {}}, {{}, {}}},
   {{{1, 2}, {1}}, {{}, {}}, {{}, {}}}, {{{1}, {}}, {{}, {}}, {{}, {}}}},
  {{{3, 2, 2, 1}, {{{{1, 4, 5}, {3, 4}}, {{}, {}}, {{}, {}}}, {{{2, 3}, {2}}, {{}, {}}, {{}, {}}},
     {{{1, 2}, {1}}, {{}, {}}, {{}, {}}}, {{{1}, {}}, {{}, {}}, {{}, {}}}}}}}
```

Code 81: applyFun(): final output

Code 81). These are the result parsing source sequence generated form *G5S* by the use of *applyFun()* and the deleting source rules.

The fact, that the in Code 81 generated sequence correspond to manually established deletion rule sequence $G4 \overset{primaryAttribute2ColumnRuleDel}{\Rightarrow} G3 \overset{subclass2TableRuleDel}{\Rightarrow} G2 \overset{subclass2TableRuleDel}{\Rightarrow} G1 \overset{class2TableRuleDel}{\Rightarrow} empty$ proves, that automated source sequence search implemented in this thesis is successful, when applied in borders of this case study.

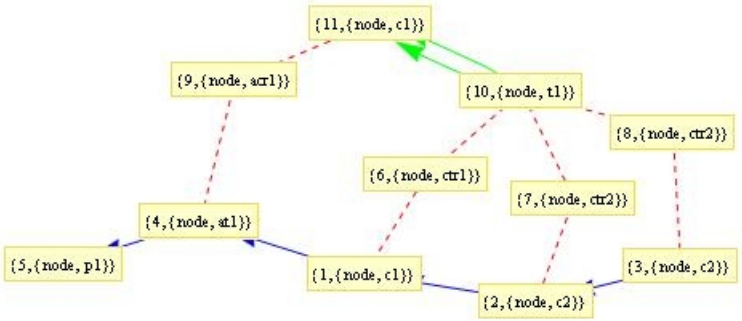
After establishing the source rule sequence finally the model transformation according to triple graph transformation theory (II. 1.3.) can be performed. Forward rules are applied to the source model *G4S* in by the triple rule source sequence defined order. This is performed by the use of *TGGapplyFWRules()(III. 3.7.)(Code 82).*

```
res = TGGapplyFWRules[Map[forwardRules[[#]] &, sourceSeq[[4]][[1]][[1]]],
  sourceSeq[[4]][[1]][[2]], G4S, SCT$TG, SCT$I]
```

Code 82: performing triple graph transformation

The automated transformation result triple graph (Code 83) represents the class diagram connected via a graph and two morphisms to the corresponding entity-relationship diagram. The fact that it corresponds with manually constructed graph *G4* from Code 76 proves, that automated triple graph transformation implemented in this thesis is successful, when applied in borders of this case study. To get the sought entity-relationship model the graph from Code 83 is reduced to the target graph(Code 84).

```
showTGG[res[[2]]]
```



Code 83: automated transformation result triple graph

2.1.2 Automated triple graph integration

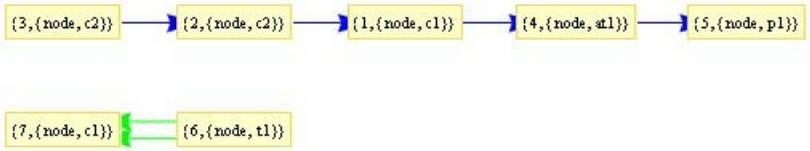
For performing an automated model integration according to the triple graph integration theory(II. 1.4.) first, like in triple graph transformation, additional methods are needed. Those methods are gained by loading "TrippleGraphIntegration.m"(Code 85) into the notebook "TripleGraphTest_classentity2_withNeeds.nb". This package contains methods like *TGGSTRules(), TGGIntegrationRules(), TGGapplyInegrationRules,* etc. These methods were developed for performing triple graph integration. In Section III. 3.8. the implementation of triple graph integration is described in detail.

```
show[res[[2]][[5]][[1]]]
```

Code 84: Code 85 ult: entity-relationship model

The task of a triple graph integration is to perform a connection between two models. For that reason, first the manually constructed test-model triple graph *G4* from Code 76 is transformed, by the use of *TGGSTGraph(),* into a source/target graph *G4ST* (Code 86).

```
G4ST = TGGSTGraph[G4, Class$TG, Entity$TG];

showTGG[G4ST]
```



Code 86: automated graph integration: source-/target- graph

G4ST represents the class and entity-relationship diagrams, which have to be connected by a third graph during the experiment automated triple graph integration in "TripleGraphT-est_classentity2_withNeeds.nb".

```
stDEL = TGGmakeDelRules[TGGSTRules[allRulesClassEntity, Class$TG, Entity$TG]];

inegrationRules = TGGIntegrationRules[allRulesClassEntity, Class$TG, Entity$TG];
```

Code 87

To perform the integration according to the triple graph integration theory (II. 1.4.) some more components are needed. The corresponding to the source-/target- graph – connection graph is build up by executing an integration rule sequence. The integration rule sequence is determined by a source-/target- rule sequence. This means, that aside from the source-/target- graph *G4ST* source-/target- and integration rules have to be generated(Code 87).

The triple graph integration theory doesn't provide any information about the generation technique of the source-/target- rule sequence. Therefore the in section III. 3.10. introduced method *applyFun()* can be used. a*pplyFun()* generates the source-/target- sequence by applying deletion source-/target- rules to the source-/target- model. That is the reason, why in *Code 87* generated source-/target- rules are transformed in corresponding deleting rules.

```
rules: {primaryAttribute2ColumnRule, subclass2TableRule, subclass2TableRule, class2TableRule}
{False, {3, 2, 2, 1},
 {{{{1, 4, 5}, {3, 4}}, {{}, {}}, {{1, 2}, {1, 2}}}, {{{2, 3}, {2}}, {{}, {}}, {{1}, {}}},
  {{{1, 2}, {1}}, {{}, {}}, {{1}, {}}}, {{{1}, {}}, {{}, {}}, {{1}, {}}}},
 {{{3, 2, 2, 1}, {{{{1, 4, 5}, {3, 4}}, {{}, {}}, {{1, 2}, {1, 2}}}, {{{2, 3}, {2}}, {{}, {}},
     {{1}, {}}}, {{{1, 2}, {1}}, {{}, {}}, {{1}, {}}}, {{{1}, {}}, {{}, {}}, {{1}, {}}}}}}}
```

Code 88: applyFun() result: source-/target- sequence

The result of *applyFun()* is a rule sequence (last output string and the second element in the output list(green underlined in Code 88)) which can be used to build up given source-/target- model. This is the result parsing source-/target- sequence, which is generated form *G4ST* by the use of *applyFun()* and the deleting source-/target- rules.

```
resST = TGGapplyInegrationRules[Map[inegrationRules[[#]] &, stSeq[[4]][[1]][[1]] ],
  stSeq[[4]][[1]][[2]], G4ST, SCT$TG, SCT$I]
```

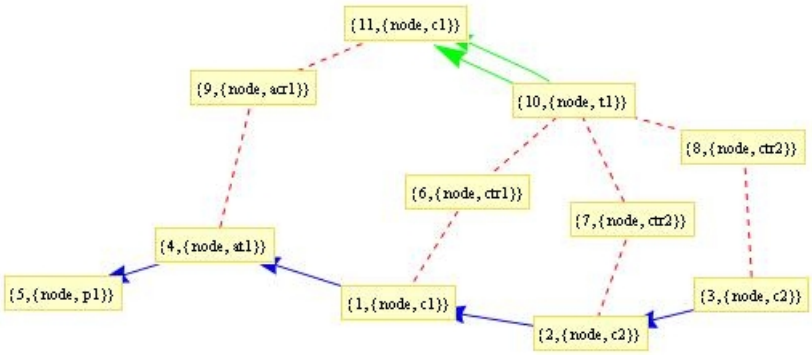
Code 89: performing triple graph integration

The fact, that the in Code 88 generated sequence correspond to manually established $G4 \overset{primaryAttribute2ColumnRuleDel}{\Rightarrow} G3 \overset{subclass2TableRuleDel}{\Rightarrow} G2 \overset{subclass2TableRuleDel}{\Rightarrow} G1 \overset{class2TableRuleDel}{\Rightarrow} empty$ deletion rule sequence proves, that automated source-/target- sequence search implemented in this thesis is successful, when applied in borders of this case study.

After establishing the source-/target- rule sequence finally the model integration according to triple graph integration theory (II. 1.4.) can be performed. Integration rules are applied to the source-/target- model *G4ST* in by the triple rule source-/target- sequence defined order. This is done by the use of *TGGapplyIntegrationRules()(III. 3.8.)(Code 89).*

The automated integration result triple graph (Code 90) represents the class diagram connected by a graph and two morphisms to the corresponding entity-relationship diagram. The fact that it corresponds with manually constructed graph *G4* from Code 76 proves, that automated triple graph integration implemented in this thesis is successful, when applied in borders of this case study.

```
showTGG[resST[[2]]]
```



Code 90: automated integration result triple graph

V. CONCLUSION

1 Future Works

1.1. ABT-Reo Case study

A third case study for model transformation / integration is already in work. It is based on the integrated model based on ABT-Reo diagrams, introduced in [BHE+10]. In this case study the model transformation / integration will be applied to triple graphs consisting of a business service model in the source component and an IT service model in the target component (Image 43). Both source and connection models are realized in ABT-Reo modelling language.

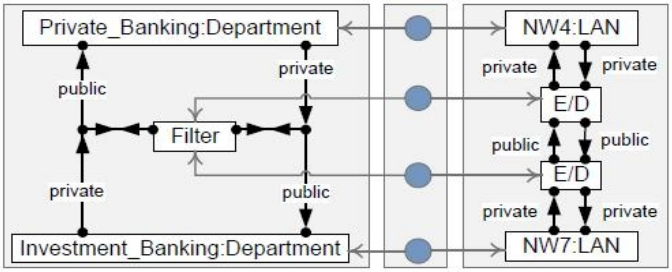


Image 43: Triple graph: business service model vs IT service model [BHE+10]

The type graph IMAGE of the case study, triple graphs and triple rules(*departmentToLan(), filterToEd(), pvpcRtL()*) required for the transformation simulated in [BHE+10] are already defined and implemented in packages: SecurityTG.m, SecurityTGRules.m, SecurityTGRules.m and SecurityTripleGraphsForRules.m.

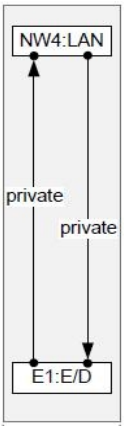


Image 45: Concrete syntax [BHE+10]

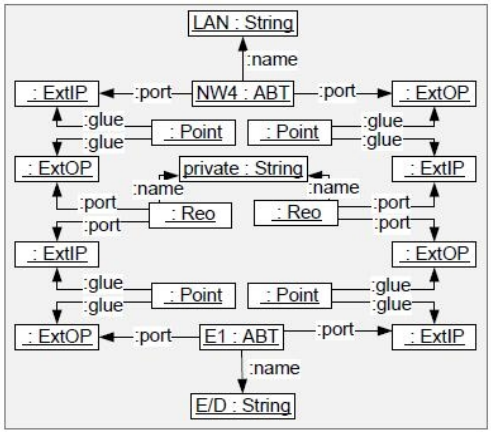


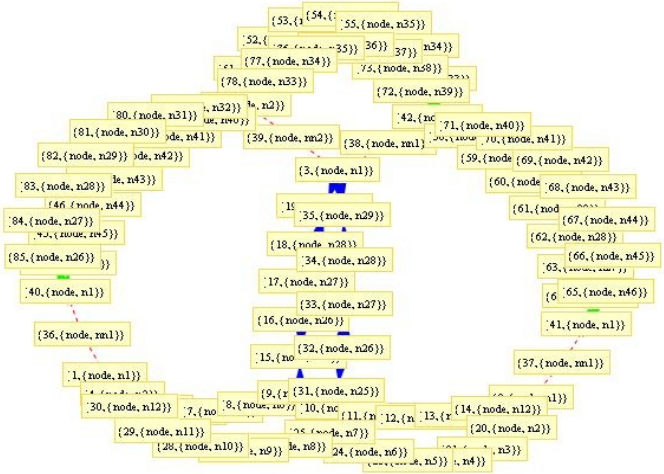
Image 44: Abstract syntax [BHE+10]

The case study evaluation is realised till a successful automated source sequence search. Yet the work on this case study had to be paused, due to leak of time and high complexity of the models, which made the execution and debugging times rise exponentially. The ABT-Reo model concrete syntax(Image 45) have a very condensed syntax. For the implementation abstract syntax is being used(Image 44 contains the abstract syntax corresponding to in Image 45 concrete syntax). This makes the generation only of the left hand triple graph source graph of the rule “private vs public”(*pvpcRtL()*) to have a confusing notation(Code 91).

```
pvpcR$L := makeTypedGraph[{{"n1", "AR$ABT:Department"}, {"n2", "AR$ExternalInputPort"},
    {"n3", "AR$Point"}, {"n4", "AR$ExternalOutputPort"}, {"n5", "AR$Reo:public"},
    {"n6", "AR$ExternalInputPort"}, {"n7", "AR$Point"}, {"n8", "AR$ExternalOutputPort"},
    {"n9", "AR$Reo:private"}, {"n10", "AR$ExternalInputPort"}, {"n11", "AR$Point"},
    {"n12", "AR$ExternalOutputPort"}, {"n13", "AR$ABT:Department"}, {"n25", "AR$ExternalInputPort"},
    {"n26", "AR$Reo:synch"}, {"n27", "AR$ExternalInputPort"}, {"n28", "AR$Point"},
    {"n29", "AR$ExternalOutputPort"}, {"n30", "AR$ABT:Filter"}},
   {{"re1", {"n1", "n2"}, "AR$ABTPort"}, {"re2", {"n3", "n2"}, "AR$glue"}, {"re3", {"n3", "n4"}, "AR$glue"},
    {"re4", {"n5", "n4"}, "AR$ABTPort"}, {"re5", {"n5", "n6"}, "AR$ABTPort"},
    {"re6", {"n7", "n6"}, "AR$glue"}, {"re7", {"n7", "n8"}, "AR$glue"}, {"re8", {"n9", "n8"}, "AR$ABTPort"},
    {"re9", {"n9", "n10"}, "AR$ABTPort"}, {"re10", {"n11", "n10"}, "AR$glue"},
    {"re11", {"n11", "n12"}, "AR$ABTPort"}, {"re12", {"n13", "n12"}, "AR$ABTPort"},
    {"re25", {"n7", "n25"}, "AR$glue"}, {"re26", {"n26", "n25"}, "AR$ABTPort"},
    {"re27", {"n26", "n27"}, "AR$ABTPort"}, {"re28", {"n28", "n27"}, "AR$glue"},
    {"re29", {"n28", "n29"}, "AR$glue"}, {"re30", {"n30", "n29"}, "AR$ABTPort"}}, AR$TG];
```

Code 91: triple rule pvpcR: left hand triple graph source graph generation

Further, because of the large amount of nodes and edges the graphical visualisation becomes also unclear. Code 92 illustrates the rule from Image 43 in abstract syntax implemented in the case study and visualised by the use of the method *showTGG()*, mentioned in chapter *IV. 2.1.1.*

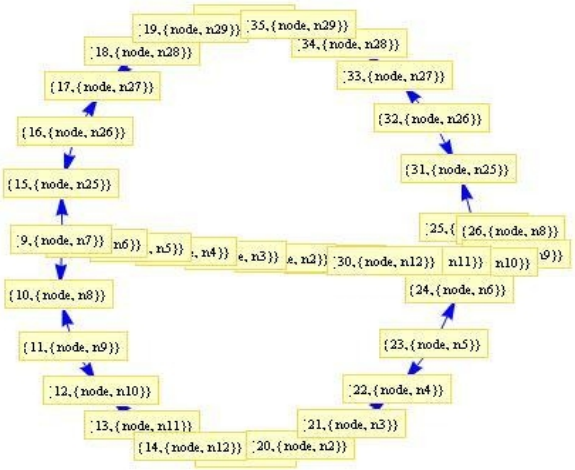


Code 92: Abstract syntax: Triple graph: business service model vs IT service model

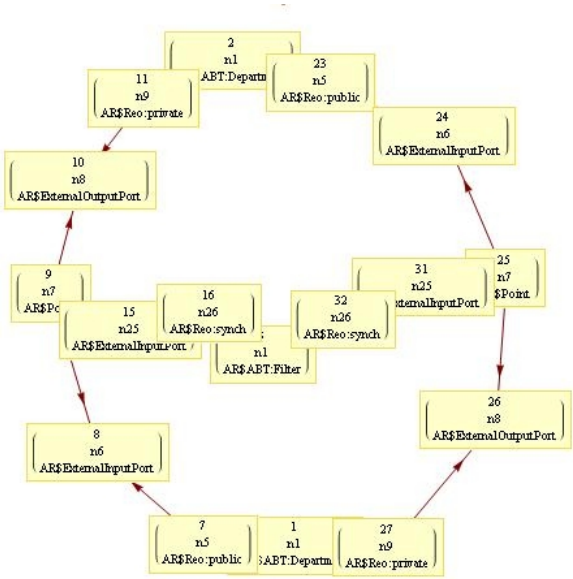
The main difficulty on the way of applying automated model transformation to the ABT-Reo case study hight calculation load in automated match search.

Therefore, before carrying on with the application of this case study. Several task a to accomplish:

1. The graph visualisation has to be extended by concrete syntax. Some progress has been already achieved in the realization of this task: *showTGSimple()* for instance visualises an in abstract syntax entered graph(Code 94) in concrete syntax(Code 93). In future it should be extended to triple graph visualisation in concrete syntax.



Code 94: business service model (Image 43 left hand side): abstract syntax



Code 93: business service model (Image 43 left hand side): concrete syntax

2. The automated match search should be made less heavy weight.(1.2.)
3. The automated source sequence search should bread a smaller search tree.(1.2.)

1.2. Future modifications for the application

Already by the application of the ABT-Reo case study it became obvious that modifications have to be performed on the match search and on the sequence search algorithms. There are several options how it can be done in future:

1. Both match search and source (source-/target-) sequence searches, contain several loops in the implementations. This loops can be replaced with list mappings. Since list mappings are Mathematica built in functions, the manipulations of lists performed by the use of this functions will provide an enormous increase of efficiency in comparison to manual iteration over a list.
2. Automated matching should filter the improper matches during the match search in the algorithms, but not after matches already have been generated.

3. The source (source-/target-) sequence search should be optimised, by limiting the deletion rules application possibilities.

All methods in the application, should throw detailed exceptions and error messages in case wrong data has been entered, or in a wrong way. Mathematica stack trace is too abstract.

1.3. Future extensions.

In first place in future several theoretical concepts should be implemented. Such as application conditions[EEP+06] and on-the-fly approach[EEE+09].

Most in this thesis implemented model transformation / integration methods already include application conditions as arguments. Yet for now this data is being ignored. In future extensions of model transformation / integration application condition should be taken into the application process of the triple rule.

The model transformation / integration theory, as it is implemented in this thesis, when applied to large projects becomes unproductive. The reason is that during the transformation / integration process a sequence of source-/forward- rules or source-/target- and integrations rule is to be build up. This process leads to constructions of huge search-trees, while building up the sequences and an enormous rise of complicity as result of extending the underlying triple rule sets. The On-the-Fly Construction (introduced in [EEE+09]) promises to fix this problem. In this approach the source (source-/target-) sequence automated generation becomes dispensable. The extension of the implementation of model transformation with On-the-Fly Construction([EEH+09]) would make a huge step towards real applicable in practice model transformation.

2 Conclusion

In this thesis, model transformation / integration based on the triple graph approach theory has been introduced and described in chapters II. 1.1. - II. 1.5. . In chapters III. 3.1. - III. 3.10. it was shown how these concepts are transformed into program code. The implementation was done in Mathematica. The usage of this programming language provided the possibility not to abstract from the theoretical concepts during the implementation. As a result an application was developed, by which models of different modelling languages can be transformed one into another, or a connection can be established between two models.

Not only the theoretical concept of triple graph model transformation was realized, but also the theory of triple graphs was extended(II. 1.5.) by the DPO rule construction. This extension provided the possibility to implement deleting triple rules(III. 3.6.1). The usage of deleting triple rules was limited to the intern usage of the application. Thanks to this extension, an efficient source(source-/target-) rules sequence pattern search has been implemented (III. 3.10.). The efficiency of the search was achieved by shrinking the parsing tree to realistic sizes.

To test the implementation, two case studies were performed. Based on entity-relationship and class diagram languages two models were specified and transformed / integrated. The case study evaluation was performed in five steps:

1. Triple rules were defined and models were build up manually.
2. Automated source rule sequence search has been performed and corresponding automated forward rule sequence has been applied to the source graph.
3. The target model of the result triple graph was compared to the manually generated one in 1..
4. Automated source-/target- rule sequence search has been performed and corresponding automated integration rule sequence has been applied to the source-/target- graph.
5. The resulting triple graph was compared to the in the manually generated one in 1..

Considering these case studies, it can be argued that the model transformation / integration implementation in this thesis is successful and correct.

A third ABT-Reo case study was started and brought to the source rule sequence generation state. Unfortunately the complexity of diagrams in this case study rose very fast. So special simplification and visualisation algorithms were developed, but the further realization of this case study is planned in borders of future works.

Bibliography

[EEH+08] Ehrig, Hartmut; Ehrig, Karsten; Herman, Frank. ***From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars. Volume 10***. 2008

[Sch94] Schürr, Andy. ***Specification of graph translators with triple graph grammars***. In Mayr and Schmidt (eds.). *Proc. WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. Berlin. 1995

[EEE+07] Ehrig, Hartmut; Ehrig, Karsten; Ermel, Claudia; Herman, Frank; Taentzer, Gabriele. ***Information Preserving Bidirectional Model Transformations***. In Dwyer and Lopes (eds.). *Fundamental Approaches to Software Engineering*. Springer. Berlin. 2007

[SE+07] S. Sumathi; S. Esakkirajan. ***Entity-Relationship Model***. *Fundamentals of Relational Database Management Systems* . Springer. Berlin. 2007

[BD+04] Bruegge, Bernd; Dutoit, Allen H.. ***Class Diagrams***. *Object-Oruented Software Engeneering* . Pearson Prentice Hall. Upper Saddle River. 2004

[EEP+06] Ehrig, Hartmut; Ehrig, Karsten; Prange, Ulrike; Taentzer, Gabriele. . . *Fundamentals of Algebraic Graph Transformation*. Springer. Berlin. 2006

[Pepper99] Pepper, Peter. . *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER* . Springer. Berlin. 1999

[MATH] ***Mathematica, Wolfram Research Homepage***. http://www.wolftam.com

[Ada09] Adamek, Jochen. ***Konzeption und Implementierung einer Anwendungsumgebung für attributierte Graphtransformation basierend auf Mathematica***. Technische Universität Berlin: Institut für Softwaretechnik und Informatik. Technical report Nr.2009/15 . 2009

[AGG] ***A Development Environment for Attributed GraphTransformation Systems***. http://user.cs.tu-berlin.de/~gragra/agg/

[BHE+10] Christoph Brandt, Frank Hermann, Hartmut Ehrig, Thomas Engel. ***Enterprise Modellingusing Algebraic Graph Transformation - Extended Version***. : . Technical report Nr. 2010/06 . 2010

[EEE+09] Ehrig, Hartmut; Ehrig, Karsten; Ermel, Claudial; Herman, Frank; Prange, Ulricke. ***On-the-Fly Construction, Correctness and Completeness of Model Transformation based on Triple Graph Grammars: Long Version***. 2009

[EEH+09] Ehrig, Hartmut; Ermel, Claudia; Herman, Frank; Prange, Ulricke. ***On-the-Fly Construction, Correctness and Completeness of Model Transformations based on Triple Graph***

Grammars: Long Version. : . Technical Report Nr. 2009/11. Fak. IV TU Berlin . 2009